The Pragmatic Programmers

TURING 图灵程序设计丛书

来自全球软件设计与定制领袖企业ThoughtWorks的宝贵创意与实践

软件开发与创新

ThoughtWorks文集（续集）

ThoughtWorks公司 著

ThoughtWorks中国公司 译

国际知名OO专家、敏捷运动创始人

Martin Fowler

以及诸多世界级程序员

与您一道：

- 应对最严苛的技术挑战
- 经营可持续的业务
- 追求软件卓越
- 推动IT变革





人民邮电出版社

POSTS & TELECOM PRESS

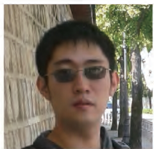
数字版权声明

主要译者简介



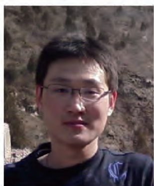
朱晓娜

曾任ThoughtWorks高级咨询师，敏捷软件开发方法实践者，北航硕士毕业。曾为多家国际知名的保险、零售、酒店管理公司、时尚内容提供商等构建企业应用，实施敏捷。



韩锴

ThoughtWorks高级咨询师，热衷于敏捷软件开发技术的实践和推广，常自诩把软件作为自己的毕生事业，却又时时被各种有趣的活动、事物和想法所吸引，最终还是回到最爱的软件上。



姜鹏

ThoughtWorks高级咨询师，有着丰富的敏捷开发经验，擅长Ruby on Rails开发。现在是内部创业产品“金数据”（http://jinshuju.net）的核心开发人员，同时也在进行敏捷与创业的实战探索。他的博客是暖风（http://jiangpeng.info）。



崔鹏飞

程序员，任职于ThoughtWorks，最爱删代码，光头迎风照三里。

TURING 图灵程序设计丛书

软件开发与创新
ThoughtWorks文集（续集）

ThoughtWorks公司 著
ThoughtWorks中国公司 译



人民邮电出版社
北京

图书在版编目（CIP）数据

软件开发与创新 ：ThoughtWorks文集 ：续集 / 美国ThoughtWorks公司著 ；ThoughtWorks中国公司译. -- 北京 ：人民邮电出版社，2014.1
（图灵程序设计丛书）
书名原文：The ThoughtWorks anthology, volume 2: More essays on software technology and innovation
ISBN 978-7-115-34294-2

Ⅰ. ①软… Ⅱ. ①美… ②T… Ⅲ. ①软件开发－文集 Ⅳ. ①TP311.52-53

中国版本图书馆CIP数据核字(2013)第315099号

内 容 提 要

本书中涵盖的软件开发主题十分广泛，从优化敏捷方法论到核心语言都有涉及。其中包括对持续集成、测试和改进软件交付过程提出的独到建议，以及如何在面向对象语言和现代 Java Web 应用程序中使用函数式编程技术等。

本书条理清晰、思维严谨却又不乏生动活泼之处，即便是书中专业性最强的文章，也不会让人觉得难以理解。除了技术人员外，本书对相关的非技术人员也很有价值。

◆ 著　　　　ThoughtWorks公司
　译　　　　ThoughtWorks中国公司
　责任编辑　丁晓昀
　责任印制　焦志炜
◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京　　　　　　　　印刷
◆ 开本：800×1000　1/16
　印张：12
　字数：283千字　　　　2014年 1 月第 1 版
　印数：1 – 4 000册　　　2014年 1 月北京第 1 次印刷
著作权合同登记号　图字：01-2013-1137号

定价：45.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

作者简介

Farooq Ali

作为一位多专业技能的通才以及T型思考者，Farooq乐于从诸多不同的角度思考问题，帮助团队提出创新性的解决方案。作为ThoughtWorks的首席咨询师，他曾经担当过许多不同的角色：开发人员、业务分析师、项目经理、用户体验设计师。Farooq对于可视化思考充满激情，并将其运用于产品构思、代码美学、数据分析等诸多领域。Farooq现任ThoughtWorks美洲社会影响力计划的负责人，为技术、创新与社会影响寻求更好的解决方案。

Ola Bini

Ola Bini来自瑞典，是一名程序设计语言极客，在ThoughtWorks芝加哥分部工作。他是JRuby的核心开发者之一，从2006年开始参与JRuby的开发。曾几何时，Ola厌倦了所有的现有语言，于是他创建了自己的语言Ioke。后来，他又厌倦了，于是又有了Seph。他曾撰写了《JRuby实战》（*Practical JRuby on Rails*）一书，还是*Using JRuby*的合著者。他在无数的会议上发言，为很多开源项目贡献过代码。他还是JSR292的专家组成员之一。

他热衷实现程序设计语言、正则表达式引擎以及寻找更好的YAML解析器的实现方式。

Brian Blignaut

Brian曾在ThoughtWorks担任了3年多的首席咨询师。在此期间，他参与交付了很多知名公司的定制项目，包括大型的面向终端客户的网站和实时流计算平台。他曾做过很多JavaScript测试方面的演讲，现在在伦敦做独立咨询师。

James Bull

James是一个有QA背景的敏捷软件开发者。他在ThoughtWorks做了很多测试自动化方面的工作。他坚信，整个团队都能从中受益的测试套件才是好的测试套件。James没在摆弄电脑的时候，肯定是在摆弄爱车。

Neal Ford

Neal Ford在ThoughtWorks担任软件架构师、Meme Wrangler。他还设计和开发过很多应用程

序，给很多杂志写过文章，做过很多视频或者DVD演讲，还曾出版、编辑、参与过8本涉及多种题材的书。他专注于设计和构建大型的企业应用程序。他还是一名在国际上颇受欢迎的演讲者，曾在300多个开发者会议上做过2000多次演讲。

他的网站是http://nealford.com，也可以通过邮件nford@thoughtworks.com联系他，他欢迎各种反馈意见。

Martin Fowler

Martin自称作家、演讲者……其实他就是一个在软件开发这个话题上喋喋不休的博学者。Martin从20世纪80年代中期就开始在软件行业工作了，也是在那时他接触到了面向对象的“新概念”。他在90年代的大多数时间是一名咨询师和培训师，帮助人们开发面向对象系统，专注于企业应用程序。他于2000年加入ThoughtWorks。

他最大的兴趣是，找出一种设计软件的方式，尽可能提高开发团队的生产率。在寻找的过程中，他努力找出好的软件设计模式，以及支持这种模式的开发流程。Martin成了敏捷方法的狂热追随者，还关注演进式软件设计。

Luca Grulla

Luca曾在ThoughtWorks工作过4年。作为首席咨询师，他帮助客户采用敏捷和精益的方法交付高质量的软件，目前在伦敦的Forward做高级程序员。他现在的工作是，试用新的语言和新的技术，每天向产品环境提交几个新特性。他还是全球IT社区的活跃成员，经常在国际性的会议上发言，还是Italian Agile Day、EuroClojure等欧洲会议的委员会成员。

Alistair Jones

Alistair Jones既是开发人员与技术负责人，又是架构师及教练。他组建出来的团队具有良好的决策能力，也能够产出优良的软件。他喜欢向人们展示，与老式的交付方式相比，敏捷方法更需要（且更能形成）严明的纪律。

Aman King

Aman King是一个应用程序开发人员。作为分布式团队的一员，他构建过很多复杂的商业应用。TDD是他的阳光和空气，他做起重构来就好像和代码有仇一样。

Patrick Kua

Patrick Kua在ThoughtWorks是一名活跃的多面手，他不喜欢被人贴上标签。Patrick多数时间都在领导技术团队，培训敏捷和精益方法。有时，他也会拯救团队于水火之中。Patrick热衷研究学习之道和持续改进，也乐于帮助他人，激起他们在这些领域的兴趣。

Marc McNeill

Marc关注把多种不同风格的团队团结起来，精诚一致地打造非凡的用户体验。Marc拥有人因工程专业的博士学位，在ThoughtWorks工作的7年间，他把设计思维和精益创业介绍给了世界各地的客户团队。他做事迅速而且注重成效，通过不断地试错帮助团队把想法转化成成功的产品。他是《当用户体验设计遇上敏捷》（*Agile Experience Design*）一书的作者之一（另一合作者是Lindsay Ratcliffe）。他的Twitter账号是@dancingmango。

Julio Maia

Julio Maia在ThoughtWorks做技术咨询师已经5年了。他在集成、自动化、运营、测试基础和应用开发等领域帮助客户解决问题，构建软件解决方案。

Mark Needham

Mark Needham是ThoughtWorks的一名软件开发人员。在ThoughtWorks工作的6年中，他使用敏捷方法帮客户解决问题，在项目中用过C#、Java、Ruby，还有Scala。

Sam Newman

Sam Newman是ThoughtWorks的技术咨询师，在ThoughtWorks工作8年有余。他在很多公司工作过，始终致力于利用技术扩展IT影响力。

Rebecca Parsons

Rebecca Parsons是ThoughtWorks的CTO，她已经想不起自己接触技术有多长时间了。她对各种技术，尤其是程序设计语言富有热情。她在莱斯大学取得了计算机科学的博士学位，其间，她专攻语言语义学和编译器。她还做过进化计算和计算生物学方面的工作。

Cosmin Stejerean

Cosmin Stejerean是有8年多经验的专业软件开发者。他现在在Simple做运维工程师，生活在得克萨斯州达拉斯市。他之前是ThoughtWorks的首席咨询师和培训师。

推　荐　序

说起做软件、写代码，不少从业人员的追求止于完成功能、解决问题。毫无疑问，这样的做法也确实能把东西做出来，但为什么渐渐地改点东西这么费劲？为什么上线交付总是磕磕碰碰？为什么用户会有那么多的抱怨？管理人员把原因归咎于流程问题、组织问题、大环境问题，开发人员则信誓旦旦地声称“只要给我足够的时间，给我足够的资源，我肯定……”。这些说法虽然也没错，然而在接触了全球众多不同的团队后，我们一个很简单的观察却是：专业的软件大多出自专业的人员，特别是当软件要解决的问题复杂到一定程度的时候。

所谓的专业人员并不仅仅是要掌握手头的编程语言、平台和工具。公司CTO Rebecca Parsons在一篇文章中诠释了ThoughtWorks对软件卓越的理解，提出了软件开发活动中应该关注的维度：业务价值、客户价值、用户体验、技术卓越、交付效力和运营效力。区别于那些介绍某种语言、工具的食谱类书籍，ThoughtWorks文集的价值在于为想要或已经成为专业从业者的人们提供了一个概要地图，并从问题域出发，着重介绍了在该领域新的实践。

ThoughtWorks文集第一卷《软件开发沉思录：ThoughtWorks文集》出版已经有4年了，在那之后，行业出现一些新的现象和趋势。多语言编程渐渐进入开发人员的视野，并在尝试中展示出一些优势；函数式编程的思想和技巧开始影响人们的编程风格；继“持续集成”之后，“持续交付”将自动化和快速反馈引向软件的全生命周期；越来越多的团队使用特性开关等手段应对快速交付中面临的诸多挑战，例如并行开发、多定制版本等；设计和创新不再是互联网和一些超炫产品公司的专属，品味日高的用户迫使各类软件团队拥抱体验设计和验证的能力。本书则捕捉并展示了这些领域的发展。

这本文集的作者是公司里在各自领域都有相当建树的同事。虽然我在公司也待了一些年头，但平时也不是有那么多的机会能够跟他们直接交流，以学习他们在项目上正在实践着的各项技术和方法。本书可以说是同事们在追求软件卓越的道路上继续前行中的又一次总结，我自己在阅读中获益匪浅。

相对第一卷散文式的组织方式，本卷则通过更加清晰的结构，更加全面地覆盖了软件开发技术和创新几个关键的方面。与此同时，仍然保持了每个章节单独成篇的特点，便于读者选择有兴趣、有需要的章节阅读。

在英文版出版之后，ThoughtWorks中国区的同事们尽快将本书翻译成了中文，方便中国的读者阅读。我衷心希望对软件开发有热情的朋友不妨一读本书，相信会对大家起到不错的借鉴和启发作用。

张松

ThoughtWorks中国区总经理，

《精益软件度量——实践者的观察与思考》作者

译 者 序

ThoughtWorker们的作品，由ThoughtWorker们翻译成中文，似乎是天经地义的。2009年，因为这个原因，我加入了第一本ThoughtWorks文集的翻译工作中；2013年，又是因为这个原因，我组织翻译了第二本ThoughtWorks文集。于是，我有幸成为唯一一名横跨两本文集的译者。

当初，我带着怎样做好软件的疑问加入了ThoughtWorks。时光荏苒，我在ThoughtWorks工作了差不多7年，当年的疑惑已经有了属于自己的答案。或许，在许多人眼中，这个答案只是“敏捷”，但我看到的是，ThoughtWorker们的精益求精。在一些人眼中已经可以接受的标准，在ThoughtWorker们眼中，却是可以做得更好：当年“日构建”的横空出世，已经让世人为之惊叹，但“持续集成”却把构建这个常见的任务提到了一个新的境界；在人们已经对持续集成甘之若饴时，ThoughtWorker们却说我们应该做“持续交付”，于是，新一轮的软件开发方式变革随之而来。持续交付会是终点吗？我不这么认为，因为我看到ThoughtWorker们仍在不断做出新的尝试。

虽然ThoughtWorker们的所有探索并非都如“持续集成”、“持续交付”般影响深远，但我们却在软件开发的各个领域进行着不断的摸索，尝试找到各种更好的解决方案。《ThoughtWorks文集》正是这样不断探索的阶段性总结。

我虽是译者，同时也是读者。单就个人喜好而言，我更喜欢第二本文集。究其原因，其一是，站在今天的角度看，第一本文集里面的有些内容稍显过时，比如，在如今这个Maven都已经不受待见的时代，谈论重构Ant构建文件显得很不合时宜；其二，纯属我作为程序员的个人偏好：第一本文集留给程序员的空间太小了，对日常开发工作有直接指导意义的内容偏少一些。而第二本文集更接地气，很多文章很程序员化——尽管它们可能不一定与许多人惯常的开发方式一致，当然，如果一致了，便也没有写出来的必要了。这些“不同寻常”的内容如果能够引发思考便是达到目的了，如果进一步改变了一些人的开发方式，那无异于对作者们的额外嘉奖。

一本由多名ThoughtWorker合作的书，由多名ThoughtWorker合作翻译，也是一件顺其自然的事。朱晓娜完成了本书第1、5、9章的翻译，韩锴负责第2、11、12章的翻译，第3、6、10章由姜鹏翻译，崔鹏飞则翻译了第4、7、8章以及序、关于作者等各种边角余料。而撰写了译者序的我，则在整个过程中打了很多零工，审校了所有的内容，统一了翻译风格。所以，但凡有任何纰漏之处，我难辞其疚。

无论在上下班拥挤的公交地铁上，还是运行构建的间隙，抑或是飘着悠扬音乐的咖啡厅里，但愿随手翻出的这本轻薄小卷，能够带给你一些思考。

祝开发愉快！

郑晔
ThoughtWorks首席咨询师，
2013 Duke选择奖获奖者

前　　言

虽说商业模式是很多公司的主要立身之本，但ThoughtWorks则是植根于一个社会模型中的。我们以三根支柱衡量公司的成败，并以此作为决策基础。

- 基业永续。
- 止于至善。
- 不为利回，不以义疚。

ThoughtWorks的这种商业模式和社会模型不断地激励着我们挑战组织结构和商业成功的现有定义。ThoughtWorks本是一个社会实验，它会随时间而演进，但我们相信100年后，ThoughtWorks仍会存在，并继续发挥着影响力。如果那时你还健在，想想书架上会有多厚的一摞ThoughtWorks文集吧！

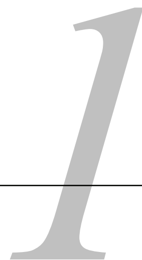
Rebecca Parsons
rjparson@thoughtworks.com
2012年6月

Martin Fowler
fowler@acm.org
2012年6月

目　　录

第四部分　数据可视化

第 1 章 引 言



Neal Ford撰文

我热爱文集。少年时，我热衷科幻小说，也有幸读到了大量内容丰富的科幻杂志。这些杂志每年都会选出最棒的故事结集成册，那可都是精华之作啊！

这些优选集伴我度过了许多悠闲时光。我之所以热爱这些文集，是因为每个故事由不同的作者写就；每每读到一个新故事，全新的写作风格都会令我耳目一新。每个故事中都有着其独到的世界观、情节设定及背景，所以，我爱文集。

多年之后，我参与编纂了多本（非虚构）文集，包括《软件开发沉思录：ThoughtWorks文集》[Inc08]，第一本ThoughtWorks文集。在变化迅速的软件行业里，博客和杂志内容分散，而专门题材的书籍又内容单一，文集恰如其分地填补了二者之间的空白。本书这样的文集如同一张时代快照，涵盖了流程、技术、哲学等诸多处于当下前沿的思想。

本书是第二本ThoughtWorks文集。编纂第一本时，Rebecca Parsons发出号召征集文章，并收到了很多高质量的投稿，最终呈现给读者的便是一本内容优秀且涉猎广泛的文集。集结第二本文集时，我们同样发出了征文号召。在此之前，大家都已听说了第一本文集，因此，第二轮征文时，大家的兴趣高了许多。我们收到了100多份稿件，其中不乏许多优秀之作。因为大家的热烈反响，我们还让ThoughtWorks技术战略委员会参与其中，这是一个辅助CTO的内部组织，协助筛选和评审这些稿件。委员会成员对提交的稿件进行了精挑细选。新的ThoughtWorks文集可谓优中选优。

正如Rebecca在本版前言中所述，ThoughtWorks是一家注重多样性的公司，包括思想的多样性。我们会在下班之后聚在一起聊聊大家又养成了哪些奇特的嗜好，或者参加午间讨论，讨论一些广泛而深远的话题，远远不只局限于软件，这些都是我们在ThoughtWorks最喜欢做的事情。我想，这些文章会让你感受到这种多样性。尽管所有文章谈的都是软件开发，但每篇文章的观点都独具特色。

正因为内容的多样性，你可以采用多种方式阅读本书。

如果你跟我一样，喜欢在不同的作者所著的全新内容之间不停切换，可以从头至尾通读本书，

当然也可以根据不同的主题进行阅读。

如果你热衷敏捷软件开发流程，请阅读第11章“交付创新”，这一章讨论了将创新力注入交付流水线的技术；你也可以从第9章“驾驭集成难题”开始阅读，其中围绕分散系统集成，讲述了解决这一棘手问题的一些行之有效的办法。

如果你想把范围缩小到敏捷与技术的交叉话题，请阅读第7章“构建更好的验收测试”、第5章“极限性能测试”以及第6章“测试驱动JavaScript”，这些文章覆盖了项目测试的方方面面。

如果你喜爱纯粹的技术话题，可以看第10章“实践中的特性开关”、第4章“使用面向对象语言进行函数式编程”、第8章“现代Java Web应用”，第3章“面向对象程序设计：对象优于类”以及第2章“最有趣的语言”。

最后，如果你相信图表的功用，那第12章“一图胜千言”将为你展示如何创建令人瞩目的可视化技术产品。

当然，阅读顺序无关紧要。这些文章是作者们利用自己的“业余”时间编写的。写作期间，他们放弃了陪伴家人和朋友的机会，也没法参加各种娱乐活动。作者对传达信息的热诚在书中展露无疑，进而也体现出他们为此所作的奉献。希望读者阅读这些文章时，能像作者写作时一样乐享其中。

Part 1

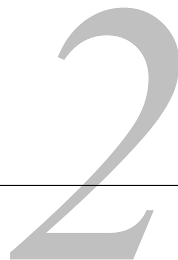
第一部分

语　言

3 位 ThoughtWorks 员工探索程序设计语言的文章，涵盖面向对象程序设计、函数式编程以及当前最有趣的语言调研。

第2章

最有趣的语言



Ola Bini撰文

编程之道

道生机器语言，机器语言生汇编器，汇编器生编译器，于今，语言逾万。

语言纵微，自有目用，皆可道软件之阴阳，皆于道中占一席之地。

但若可能，远离COBOL。

一场语言的复兴正在酝酿之中，而且已经持续了很多年。对于语言极客来说，当下的生活可能是自20世纪70年代以来最好的年代了。我们亲历了很多新语言的诞生，也看到很多古老的语言在新的领域里重焕生机——或者像Erlang一样，它所能解决的问题在今天突然变得至关重要。

为什么我们今天会看到语言的复兴？很大一部分原因在于我们正在面对更艰难的问题。代码库一天比一天臃肿，传统的工作方法不再有效。我们在越来越紧迫的时间压力下工作——尤其是创业公司，其生死就取决于产品的发布速度。另外，我们面对的问题越来越依赖并发与并行能力。面对这些问题，传统方法早已过时，因此，诸多开发者转向不同的语言，希望新的语言能以更简单的方式解决问题。

一方面，我们对新方法的渴求正与日俱增，而另一方面，我们也拥有了更加丰富的资源，可以创建新的语言。创建语言所需的工具已经达到了全新的高度，甚至在几天之内就可以打造出一个可用的语言。一旦有了可运行的语言，我们就可以把它放到任何一个成熟的平台（比如JVM、CLR或者LLVM）上运行。如果语言运行在这些平台上，它就可以访问平台上的库、框架和工具，也正是这些东西都让这些平台如此强大。这也意味着，语言的创造者不必重新发明“轮子”。

本文将讨论当下一些有趣的语言。我相信每个程序员都能从本文中列举的语言中有所斩获。不过这样的语言清单必然是主观的，且会随时间而改变。我希望在未来几年之内，这些语言都不会过时。

2.1 为什么语言很重要

构建计算机科学的基础之一是邱奇–图灵命题（Church-Turing Thesis）。该命题及其相关结论有效地证明了，不同的编程语言在基本层面上并没有区别。你用一种语言可以实现的事情，用另一种语言同样可以。

既然如此，我们又何必在意编程语言之间的区别呢？为什么你不继续用Java写程序呢？想想吧，如果语言真的无关紧要，为什么有人会发明Java，又为什么会有人用它呢？玩笑归玩笑，这里的重点是，我们很在乎选择编程语言，而理由很简单，就是这些语言各有所长。即使能够使用任何语言做任何事情，但很多情况下，在一种语言里做某事的最好方式却等价于创建了另一种语言的解释器。这有时会被称为格林斯潘编程第十定律（Greenspun's Tenth Rule of Programming）：

> 任何C或Fortran程序复杂到一定程度之后，都会包含一个临时的、只有一半功能的、不完全符合规格的、到处者是bug的、运行速度很慢的Common Lisp实现。

事实上大多数语言都可以完成大部分工作，区别只是在于实现起来的难易程度。因此，为某项任务选择一种正确的语言，就意味着接下来所有工作都会变得更为轻松。而了解多种语言，也意味着你在解决特定问题时能有更多的选择。

作为程序员，我认为使用哪种编程语言是最重要的选择。语言的选择需要慎重，因为其他一切工作都依赖这门语言，甚至可以说语言的选择是项目存亡的关键。

2.2 一些有趣的语言

我知道众口难调，也无法在一篇文章中满足所有人的喜好。如果你所钟爱的语言没有列出来，并不意味着我认为它无趣。我考察了许多语言，却由于篇幅限制，无法一一展示，最后只好甄选出其中几种列在本文中。它们最能体现出不同类型语言之间的差异。有兴趣的话，你也可以像我一样甄选出其他的语言。如果你没看到自己最爱的语言出现在这里，请不要失望，可以写邮件告诉我为什么你认为某种语言应该在本文列出。或者你也可以写一篇相同模式的博客，介绍你所喜欢的有趣的语言。

本文不会介绍如何下载和安装文中提到的语言。此类内容变化很快，因此你最好求助Google。我也不会为你展示某种语言的所有内容，只是介绍一些值得注意的特性，希望以此引起你对它们的兴趣。

2.2.1 Clojure

Rich Hickey于2007年发布了第一版的Clojure。从那时起，Clojure开始迅速流行起来。现在，

Clojure背后已有商业化运作，募集了大量开发资金，并且有几本非常不错的相关书籍。Clojure本身发展得也很快——自从第一版发布以来，已经有4个重要版本发布：1.0，1.1，1.2和1.3[①]。这4个版本给Clojure带来了极大的提高和改善。

Clojure是一种Lisp方言，但却并不是Common Lisp或Scheme的某种实现。事实上，它从众多不同的语言中汲取了很多灵感，因此是一个全新版本的Lisp。它运行于JVM之上，可以轻松访问任何既有的Java程序库。

有Lisp编程经验的读者一定知道“列表”（list）是Lisp的核心。Clojure将此特点发扬光大，并在列表之上加入了额外的抽象。因此，数据结构成为了该语言的核心。不仅是列表，还有向量、集合、Map，所有这些数据结构都有相应的语法，Clojure程序的代码本质上即是由这些数据结构写就的，也最终会表现为这些数据结构。相比于其他语言，Clojure有一点不同，它的数据结构是不能修改的。如果要修改它们，首先要描述出这个修改是什么样的，然后就会返回一个新的数据结构，旧的那个依然存在并可以使用。这似乎过于浪费。的确如此，它确实不如直接摆弄字节码效率高。但它也不像你想象的那样慢，因为Clojure的数据结构实现是非常成熟和智能的。刚才提到的数据结构的不变性，使Clojure可以轻松完成很多其他语言看来非常困难的工作。不可变的数据结构有一个非常明显的好处：由于永远不能修改，因而肯定能保证它们满足线程安全的要求。

今天，我们选择Clojure的主要原因在于它有一套非常周密的模式，十分适合处理并行与并发执行。Clojure的基本特点是不可变性。如果你需要可变性，那么可以根据控制可变性的预期方式，创建一些特殊的数据结构来进行模拟。

比如说，你想要确保某3个变量能同时改变，那么在Clojure里做到这一点并不难，只要将这些变量存入几个引用`ref`，然后使用Clojure的软件事务性存储（STM）来协调变量访问即可。

总之，Clojure有很多不错的特性。它与Java的互操作性非常实用。我们可以完全控制程序中的并发行为，同时不必使用诸如锁、互斥量（mutex）等容易出错的方法。

接下来，我们看看Clojure的代码。第一个例子是简单的“Hello World”程序。和很多所谓的脚本语言一样，Clojure在顶层就可以执行任何东西。下述代码首先定义了一个函数，名为`(hello)`，然后传入两个不同的参数调用它。

MostInterestingLanguages/clojure/hello.clj

```
(defn hello [name]
  (println "Hello" name))

(hello "Ola")
(hello "Stella")
```

① 截止翻译时，Clojure 1.5已经发布。——译者注

如果定义了clj命令，我们可以运行这个文件获得以下预期结果：

```
$ clj hello.clj
Hello Ola
Hello Stella
```


前文提到过，在Clojure中使用数据结构非常方便，而且它们也提供了很强大的操作。下面的例子示范了如何创建不同的数据结构，并获取它们的元素。

MostInterestingLanguages/clojure/data_structures.clj

```
(def a_value 42)

(def a_list '(55 24 10))

(def a_vector [1 1 2 3 5])

(def a_map {:one 1 :two 2 :three 3, :four 4})

(def a_set #{1 2 3})

(println (first a_list))
(println (nth a_vector 4))
(println (:three a_map))
(println (contains? a_set 3))

(let [[x y z] a_list]
  (println x)
  (println y)
  (println z))
```

这段代码的最后几行所做的事情是最有意思的。let语句可以把集合解构成几部分，本例只是拆分了一个含有3个元素的列表，然后分别赋给x、y和z。事实上，Clojure可以任意嵌套和解构这样的集合。

运行上面的代码，将得到如下输出：

```
$ clj data_structures.clj
55
5
3
true
55
24
10
```

使用Clojure的数据容器时，通常都是这样添加或移除元素的，继而可以创建新容器。无论使用什么容器，Clojure都会支持3个函数，满足你的大部分需要。这些函数是(count)、(conj)和(seq)。(count)函数不言自明。调用容器的(conj)函数可以向容器中添加新内容，不过，到

底插在哪取决于容器的类型，比如使用(conj)向List中加入元素，新元素会在List的头部。对Vector来说，新元素则会在尾部，而对Map而言，(conj)会添加一个键/值对。

为了支持数据容器上的一些更常用的操作，Clojure提供了一个名为Sequence的抽象。调用(seq)可以将任何一个集合转化为Sequence。一旦有了Sequence，就可以使用(first)和(rest)遍历它了。

那么，这在实践中怎么用呢？

MostInterestingLanguages/clojure/data_structures2.clj

```
(def a_list '(1 2 3 4))
(def a_map {:foo 42 :bar 12})

(println (first a_list))
(println (rest a_list))

(println (first a_map))
(println (rest a_map))

(def another_map (conj a_map [:quux 32]))

(println a_map)
(println another_map)
```

在这段代码中，先分别打印了一个list，map的第一个元素以及剩余的部分。然后，向现有的map中添加了一个键/值对，从而创建新的map，而原来的那个map保持不变。执行这段代码，输出如下：

```
$ clj data_structures2.clj
1
(2 3 4)
[:foo 42]
([:bar 12])
{:foo 42, :bar 12}
{:foo 42, :quux 32, :bar 12}
```

Clojure非常容易和Java集成。事实上，有时候很难分辨出Clojure与Java代码的界线。比如，我们前面谈到的Sequence抽象机制。它其实就是一个Java接口。Clojure与Java库的互操作通常就是直接调用Java库。

MostInterestingLanguages/clojure/java_interop.clj

```
(def a_hash_map (new java.util.HashMap))
(def a_tree_map (java.util.TreeMap.))

(println a_hash_map)
(.put a_hash_map "foo" "42")
```

```
(.put a_hash_map "bar" "46")

(println a_hash_map)
(println (first a_hash_map))
(println (.toUpperCase "hello"))
```

任何在Classpath上的Java类都能很容易地实例化：可以调用(new)，并将class作为参数传入；还有一种特殊形式——在类名后面加一个点（.），把它当作一个函数。获得了Java实例后，可以像使用任何Clojure对象一样使用Java对象。调用对象上的Java方法需要特殊的语法，即在方法名前面加一个点。以这种方式调用Java方法并不局限于通过Java类创建的对象。实际上，Clojure的字符串只是普通的Java字符串，所以，我们可以直接对它调用toUpperCaes()。

执行上述代码，将得到下面的输出：

```
$ clj java_interop.clj
#<HashMap {}>
#<HashMap {foo=42, bar=46}>
#<Entry foo=42>
HELLO
```

前面提到了Clojure的并发问题，因此，我想演示一下STM的使用。虽然这听上去非常高深，但实际中用起来却相当简单。

MostInterestingLanguages/clojure/stm.clj

```
(defn transfer [from to amount]
  (dosync
   (alter from #(- % amount))
   (alter to #(+ % amount))
   )
  )

(def ola_balance (ref 42))
(def matt_balance (ref 4000))

(println @ola_balance @matt_balance)

(transfer matt_balance ola_balance 200)

(println @ola_balance @matt_balance)

(transfer ola_balance matt_balance 2)

(println @ola_balance @matt_balance)
```

这个例子做了很多事情，不过需要注意的是(ref)、(dosync)和(alter)。在代码中调用(ref)将创建一个引用，并赋给它初始值。@符号用来取出引用的当前值。(dosync)代码块中执行的任何操作都会在一个事务中完成，这就意味着没有任何代码可以看到(dosync)代码块处于不一致的状态。

然而，为了让这种情况成为可能，`(dosync)`可能会多次执行其代码。`(alter)`会真正改变引用的值。井号（`#`）语法在Clojure中用于创建匿名函数，这种语法非常独特。

运行这段代码将会得到预期的输出。这段代码其实没有使用任何线程，不过如果真有很多线程需要交错使用这些引用的话，也不必担心结果的正确性。

```
$ clj stm.clj
42 4000
242 3800
240 3802
```

我本想在这一节中向你展示更多的Clojure特性，不过此刻我们必须开始下一个语言了。如果想获得更多关于Clojure的信息，可以看一看下面的学习资源。我强烈建议你尝试下Clojure——那的确是一种令人愉悦的体验。

学习资源

关于Clojure的书已经有不少了。Stuart Halloway的*Programming Clojure* [Hal09]是第一本关于Clojure的书，而且至今仍然可以用这本书来了解Clojure。这本书的第二版刚刚发布，并且多了一位合著者——Aaron Bedra。

同时，我也是*The Joy of Clojure*[FH11]一书的粉丝，它会教你如何写出地道规范的Clojure代码。

对语言有了全面的了解后，我推荐你看看Clojure的主页（http://clojure.org）。其中有很多不错的资源，通过阅读该网站的文章，你能学到很多关于Clojure的知识。

最后，Clojure的邮件列表是学习过程中的重要辅助。这是个非常活跃的列表，你经常会看到Clojure的核心成员回答问题。这里也经常会探讨Clojure未来的新特性。

2.2.2　CoffeeScript

在过去几年里，JavaScript迅速流行起来。其中的主要原因在于，越来越多的公司选择HTML5作为应用程序的主要发布机制；此外，对于Web应用程序来说，创建更好的用户接口也越来越关键。基于上述原因，我们用JavaScript写的程序越来越多。但是这里有一个大问题，JavaScript有时很难写好。它有一个奇特的对象模型，而且工作方式表面上看来总是不那么合理，语法也非常笨重。

于是就有了CoffeeScript。

CoffeeScript是一种相对比较新的语言，不过，GitHub上的排名说明它已经是一个最有趣的项目了。在这个语言集合中，它还是一个“不合群的家伙”，因为它还算不上一门完整的语言。它更像JavaScript之上薄薄的一层——它能编译为可读性相当好的JavaScript代码。它从Ruby和Python中获得了大量灵感，如果用过其中任何一种语言，你都会觉得CoffeeScript非常顺手。

就和Python一样，Coffee使用缩进组织程序的结构。它的主要目标是要比JavaScript更具可读性和易用性，这在很大程度上是指语法层面上的。

尽管语法是很重要的因素，但CoffeeScript并不仅仅是语法不同。它也支持其他高级特性，比如推导（comprehension）和模式匹配。

CoffeeScript还很容易建立基于类的继承结构，它有特定的语法支持。JavaScript中更令人苦恼的一点，就是有了一个正确的继承结构后，如何把它们组织到一起？如果你刚刚从某个标准的基于类的面向对象的系统转过来，那么你就会发现CoffeeScript更容易接受。

到目前为止，CoffeeScript并没有任何主要的新功能，但是它能使应用程序的JavaScript部分更简单。它还能使JavaScript代码更一致，容易阅读和维护。

那我们就开始吧！看看“Hello World”的CoffeeScript代码：

MostInterestingLanguages/coffee_script/hello.coffee

```
greeting = "hello: "
hello = (name) =>
    console.log greeting + name

hello "Ola"
hello "Stella"
```

如果你安装了CoffeeScript和Node.js，可以这样运行：

```
$ coffee hello.coffee
hello: Ola
hello: Stella
```

通过这个简单例子可以看到，我们创建的方法是一个词法闭包，使用的变量是`greeting`。和Ruby一样，不需要括号。解析器尽量简化这部分工作。

在CoffeeScript中创建内嵌对象是非常容易的。既可以使用显式的分割符，也可以使用缩进表明对象的起始与终止。

MostInterestingLanguages/coffee_script/nested_objects.coffee

```
words = ["foo", "bar", "quux"]
numbers = {One: 1, Three: 3, Four: 4}

sudoku = [
    4, 3, 5
    6, 8, 2
    1, 9, 7
]

languages =
    ruby:
```

```
        creator: "Matz"
        appeared: 1995

    clojure:
        creator: "Rich Hickey"
        appeared: 2007

console.log words
console.log numbers
console.log sudoku
console.log languages
```

运行这段代码，将得到如下输出：

```
$ coffee nested_objects.coffee
[ 'foo', 'bar', 'quux' ]
{ One: 1, Three: 3, Four: 4 }
[ 4, 3, 5, 6, 8, 2, 1, 9, 7 ]
{ ruby: { creator: 'Matz', appeared: 1995 }
, clojure: { creator: 'Rich Hickey', appeared: 2007 }
}
```

打印的输出有点儿乱，不像创建语句那么干净，对此我稍感遗憾。不过我想这正是CoffeeScript的一个优势——可以非常干净地创建内嵌对象。

CoffeeScript的另一个优势在于可以使用`for`关键字定义对象的推导：

MostInterestingLanguages/coffee_script/comprehensions.coffee

```
values =
    for x in [1..100] by 2 when 1000 < x*x*x < 10000
        [x, x*x*x]

console.log values
```

运行这段代码，可以得到所有在1至100之间，且立方值在1000至10000之间的奇数。

```
$ coffee comprehensions.coffee
[ [ 11, 1331 ]
, [ 13, 2197 ]
, [ 15, 3375 ]
, [ 17, 4913 ]
, [ 19, 6859 ]
, [ 21, 9261 ]
]
```

CoffeeScript的推导不仅可以结合`list`和`range`完成很多与数据容器相关的操作，也可以与`object`和`dictionary`很好地配合。

MostInterestingLanguages/coffee_script/classes.coffee

```
class Component
    constructor: (@name) ->

    print: ->
        console.log "component #{@name}"

class Label extends Component
    constructor: (@title) ->
        super "Label: #{@title}"

    print: ->
        console.log @title

class Composite extends Component
    constructor: (@objects...) ->
        super "composite"
    print: ->
        console.log "["
        object.print() for object in @objects
        console.log "]"

l1 = new Label "hello"
l2 = new Label "goodbye"
l3 = new Label "42"

new Composite(l1, l3, l2).print()
```

从上面这最后一个例子可以看出，如果我们的问题更适合用传统的面向对象的结构表达，CoffeeScript也是可以办到的。如果熟悉Java或者Ruby的规律，那么使用CoffeeScript中的构造函数和`super`就会非常顺手。这段程序的输出结果为：

```
$ coffee classes.coffee
[
hello
42
goodbye
]
```

如果你以前用过但不喜欢JavaScript，CoffeeScript则是个不错的代替品。它可以同时应用于服务器端和客户端。Rails现在已经绑定了CoffeeScript。你不应该错过！

学习资源

学习CoffeeScript的最佳起点是http://coffeescript.org。该站点有一份语言特性汇总，内容清晰明确。此外，这个站点还包含了一个可交互的控制台，可以在上面键入CoffeeScript代码，然后马上可以生成翻译好的JavaScript代码。

Trevor Burnham的著作《深入浅出CoffeeScript》（*CoffeeScript* [Bur11]）也是不错的资源。

如果你喜欢通过示例学习编程语言，CoffeeScript的主页上还有一些注释过的源码，供人了解其内部实现。这会让阅读和理解代码程序更加容易。

2.2.3 Erlang

Erlang是本文中最古老的语言，约诞生于20世纪80年代。不过，直到最近人们才开始真正关注它。

Erlang最初由Joe Armstrong开发，当时是为了编写可容错的程序。Erlang主要应用的领域是长距离电话交换机以及其他相关领域。这些领域的最关键因素在于系统的正常运行时间。Erlang的大部分需求都来自于对代码的要求：健壮、容错、可以在运行时期更换。

今天，Erlang正越来越多地应用于其他领域，其原因在于它底层的Actor模型非常适合创建健壮的、可扩展的服务。

Erlang是函数式语言。函数是"一等公民"，可以在需要的时候创建，可作为参数传递，还可作为其他函数的返回值。Erlang只允许给一个变量名赋值一次——因此实现了不可变性。

Erlang的核心模型是Actor模型。其思想是我们可以拥有大量轻量的进程（称为Actor），它们可以通过发送消息来彼此通信。所以在Erlang中，要利用Actor来对行为建模或修改状态。如果你已经在其他语言中使用过进程和线程，一定要记住Erlang的进程是非常不同的：它们是轻量的，可以快速地创建，而且可以按照需要分布到不同的物理机器上。这样我们写出的同一份代码，既可以在一台机器上运行，也可以在上百台机器上运行。

与Erlang紧密相连的是开放电信平台（Open Telecom Platform，OTP），它是一些库的集合，可以用来创建非常健壮的服务。该平台给程序员提供了一个框架，程序员可以通过钩子使用一些高级的模式，创建可靠的Erlang服务——比如让特定Actor监控其他Actor的健康状况、Actor在运行期间可以实现代码的热替换等其他一些高级特性。

和前面看到过的语言不同，Erlang无法从脚本的顶层直接运行，因此在例子中，我们会从Erlang控制台中执行代码。但这样会产生的一个副作用，即使最简单的程序也会比较长，因为我们必须将它作为一个Erlang模块导出。

MostInterestingLanguages/erlang/hello.erl

```
-module(hello).
-export([hello/1]).

hello(Name) ->
    io:format("Hello ~s~n", [Name]).
```

前两行代码是导出模块信息的指令。我们定义了一个hello()函数。变量名必须以大写字母开头，比如Name就是参数变量。format()是io模块中的函数。Erlang可以非常灵活地进行格式化。在这段程序中，我们只是把名字插入到字符串，然后打印出来。

在Erlang shell中执行这段代码，可以看到：

```
1> c(hello).
{ok,hello}
2> hello:hello("Ola").
Hello Ola
ok
3> hello:hello("Stella").
Hello Stella
ok
```

每个Erlang语句都以句点（.）结束，告诉解释器输入已完成。我们必须先编译模块，然后才能使用它。c()函数用来完成模块的编译。然后，我们可以调用模块。值ok是我们创建的函数的返回值。

虽然Erlang是函数式语言，不过它真正的强项是支持模式匹配和递归算法。在看下一个例子之前，首先要知道，Erlang中的小写字母开头的标识符是符号，花括号括起来的是元组（tuple），方括号括起来的是列表。在Erlang中，这三者的组合可以表现不同类型的事物，通常是模式匹配的对象。

MostInterestingLanguages/erlang/patterns.erl

```
-module(patterns).
-export([run/0]).

run() ->
    io:format("- ~s~n", [pattern_in_func("something")]),
    io:format("- ~w~n", [pattern_in_func({foo, 43})]),
    io:format("- ~w~n", [pattern_in_func({foo, 42})]),
    io:format("- ~s~n", [pattern_in_func([])]),
    io:format("- ~s~n", [pattern_in_func(["foo"])]),
    io:format("- ~s~n", [pattern_in_func(["foo", "bar"])]),
    io:format("- ~w~n", [pattern_in_case()]),
    io:format("- ~w~n", [reverse([1,2,3])]),
    io:format("- ~w~n", [reverse([])])
    .

pattern_in_func({foo, 43}) ->
    23;
pattern_in_func({foo, Value}) ->
    Value + 10;
pattern_in_func([]) ->
    "Empty list";
pattern_in_func([H|[]]) ->
```

```
        "List with one element";
pattern_in_func(X) ->
        "Something else".

pattern_in_case() ->
        case {42, [55, 60]} of
            {55, [42 | Rest]} -> {rest, Rest};
            {42, [55 | Rest]} -> {something, Rest}
            end.
reverse(L) ->
        reverse(L, []).

reverse([], Accum) ->
        Accum;

reverse([H|T], Accum) ->
        reverse(T, [H] ++ Accum).
```

这段代码首先创建了一个`run()`方法，它会执行定义在模块中的其他方法。Erlang的模式匹配用于三种场景，第一种是函数参数，第二种是`case`语句，第三种是用于处理消息传递。这段代码只示范了前两种情况。此外，代码还示范了如何使用Erlang的模式匹配机制，轻松实现尾递归算法。

```
18> c(patterns).
{ok,patterns}
19> patterns:run().
- Something else
- 23
- 52
- Empty list
- List with one element
- Something else
- {something,[60]}
- [3,2,1]
- []
ok
```

我们在列表中使用了管道（|）语法，把列表的头元素和剩余的元素分开。这种拆分数据集，然后分别对头、尾部分进行处理的模式是很多函数式语言中常见的模式。在`reverse()`函数的例子中，我只是简单地把头元素和尾部调换顺序，再重新组装在一起。

Erlang广为人知的一点是它支持Actor。在下一个例子中，我们会看到一个Actor，它将包含几种状态。某种程度上说，Actor和一个能永远保持内部一致性的同步内存区域相似。这里需要理解的主要语法是叹号（!），它用于向Actor发送消息。我们可以把任何可序列化的Erlang表达式发给Actor，甚至发送一个函数。`receive`关键字此时的用处更像一个`case`语句，不同之处在于它会等待发送给当前正在运行的Actor的消息。

MostInterestingLanguages/erlang/actor.erl

```
-module(actor).
-export([run/0]).

run() ->
    State1 = spawn(fun() -> state(42) end),
    State2 = spawn(fun() -> state(2000) end),
    io:format("State1 ~w~n", [get_from(State1)]),
    io:format("State2 ~w~n", [get_from(State2)]),

    State1 ! {inc}, State1 ! {inc},
    State2 ! {inc}, State2 ! {inc}, State2 ! {inc},

    io:format("State1 ~w~n", [get_from(State1)]),
    io:format("State2 ~w~n", [get_from(State2)]),

    State1 ! {update, fun(Value) -> Value * 100 end},

    io:format("State1 ~w~n", [get_from(State1)]),
    io:format("State2 ~w~n", [get_from(State2)])
    .

get_from(State) ->
    State ! {self(), get},
    receive
        Value ->
            Value
    end.

state(Value) ->
    receive
        {From, get} ->
            From ! Value,
            state(Value);
        {inc} ->
            state(Value + 1);
        {From, cas, OldValue, NewValue} ->
            case Value of
                {OldValue} ->
                    From ! {set, NewValue},
                    state(NewValue);
                _ ->
                    From ! {notset, Value},
                    state(Value)
            end;
        {update, Func} ->
            state(Func(Value))
        end.
```

这段代码中定义了三个不同的函数。第一个用来运行整个示例。它调用了两次`spawn`，创建了两个不同的状态Actor。简单地说，一个Actor就是一个运行中的函数，所以代码中使用`fun`关键字来创建匿名函数，并且分别设置初始值42和2000。获得初始值后，首先打印它们，接着将第一个状态自增2次，第二个状态自增3次，然后再次打印它们，最后再向Actor发送一个函数，该函数可以用原始值乘以100后得出一个新值。结束前，再一次打印所有的值。第二个函数是`get_from()`，它是一个助手函数，可以简化从Actor获取值的过程。它会向Actor发送一个`get`消息（Actor来自于`get_from`的参数），然后等待接收答案。

最后的函数是个真正的Actor。它会等待消息，然后根据收到的不同消息作出不同的响应。`state`执行结束后要递归地调用自身，以此保持状态。

```
32> c(actor).
{ok,actor}
33> actor:run().
State1 42
State2 2000
State1 44
State2 2003
State1 4400
State2 2003
ok
```

如果你可能需要花些时间才能完全理解最后一个例子，也不用太着急。因为Erlang处理状态的方式和大多数编程语言差别很大。总的来说，Erlang提供了非常强大的原语来处理并发，而且，巧妙地组合运用Actor可以写出非常漂亮的算法。

学习资源

学习Erlang的最佳起点就是Joe Armstrong的《Erlang程序设计》（*Programming Erlang* [Arm07]）。该书对于Erlang的各个方面都有完整的介绍，对其中较为复杂的部分均有涉猎。此外，还有一本不错的书是Francesco Cesarini和Simon Thompson所著的*Erlang Programming* [CT09]。

你也可以网上获得一些不错的资源，比如http://learnyousomeerlang.com。

2.2.4　Factor

Factor创建于2003年，其灵感源自更加古老的语言Forth。它是面向栈的编程语言，因此其编程模型与大多数程序员所熟悉的大相径庭。随着Factor的发展变化，其使用方式也渐渐地产生了极大的变化。Factor曾经是基于JVM的，不过，现在大部分已由自身实现，可以运行在所有主流的平台上。

基于栈的语言的编程模型看上去非常简单。所有任务都在栈上。每个操作都从栈上获取值，并把结果存到栈上，大部分时间这些都是隐式完成的。比如，两个数的加法运算，首先将两个数推入栈顶，然后执行`plus()`指令。该指令从栈顶获取两个操作数，然后把结果再推回到栈顶。

基于栈的语言中，很多使用栈的地方在其他语言中都是用变量完成的。大多数情况下，基于栈的语言也通过栈给函数传递参数。

Factor的标准发行版中包含了大量的标准类库。而且该语言自身也包含了很多高级特性，比如类型系统、Tuple类、宏、用户自定义的解析词汇（parsing word），以及强大的外部语言接口。

Factor的语法使用逆波兰表示法（Reverse Polish Notation，RPN），非常简单。虽然该方法需要花些时间来掌握，不过一旦适应了，就会觉得非常自然，我们可以显式地遵循栈上的操作。

MostInterestingLanguages/factor/hello.factor

```
USE: io
USE: kernel
USE: sequences
IN: user

: hello ( x -- ) "hello " swap append print ;

"Ola" hello
"Stella" hello
```

这段代码的功能和前面看到的“Hello World”无异。我们首先定义要在程序中使用的模块，然后通过`IN: user`声明我们处在用户词汇表（User vocabulary）中。接下来的一行以冒号开头，定义了一个新词汇`hello()`。在括号里，我们指明`hello`对栈的影响只是从中取出一个元素，之后不会再往栈中放置任何东西。最后，我们将字符串`hello`推入栈，然后交换栈顶两个字符串的位置，把它们拼接到一起，最后打印结果。在定义了`hello`之后，我们可以往栈里再压入一个字符串，然后调用`hello`。

如果你能在命令行运行Factor，那么运行结果如下：

```
$ factor hello.factor
hello Ola
hello Stella
```

因为你需要时刻追踪栈的当前状态，因此思考Factor代码的方式是有本质不同的。而且你必须确保栈上的内容从下到上都是正确的。词汇要定义栈的输入和输出，主要原因在于如果程序的最终结果与期望不符，Factor不会接受这样的程序。

下面的例子示范了几个小程序，说明了Factor能够轻松解决的问题：

MostInterestingLanguages/factor/parsing_passwd.factor

```
USE: io
USE: io.encodings.utf8
USE: io.files
USE: kernel
USE: math
```

```
USE: sequences
USE: splitting
USE: xml.syntax
USE: xml.writer
IN: user

3 [ "Hello" print ] times

{ "foo" "bar" "baz" }
[ [XML <li><-></li> XML] ] map
[XML <ul><-></ul> XML] pprint-xml

nl nl

: separate-lines ( seq -- seq2 ) [ ":" split first ] map ;
: remove-comments ( seq -- seq2 ) [ "#" head? not ] filter ;
: remove-underscore-names ( seq -- seq2 ) [ "_" head? not ] filter ;

"/etc/passwd" utf8 file-lines
     separate-lines remove-comments remove-underscore-names
     [ print ] each
```

第一段程序（`use`语句之后）在控制台上打印三遍`hello`。它首先将数字3和一个所谓的quotation压入栈。Factor的quotation基本上类似匿名函数。在本例中，quotation的任务只是打印hello，不过它也可以使用栈上的值，或者将值压回栈里（产生副作用）。最后调用`times()`词汇，它会执行三次quotation中的代码块。

第二段程序示范了Factor的一个强大的功能——我们可以创建自己的解析器词汇定义特定的语法。Factor已经包含了很多不同的解析器。本例展示了XML字面的语法。然而，这个语法没有内置在语言之中；它是被定义为一个库。在这段程序中，我首先往栈中压入包含三个元素的列表，然后使用`map()`依次创建XML的片段，最后使用`pprint-xml()`将它以漂亮的格式打印出来。

最后一段代码首先定义了几个助手词汇，分别是`separate-lines()`、`remove-comments()`和`remove-underscore-names()`。它们用于读取/ect/passwd文件中的所有行，分出所有不同的列，然后只保留以下划线开始的用户名，最后打印结果。

执行这段程序，将得到如下输出——取决于/etc/passwd文件内容：

```
$ factor parsing_passwd.factor
Hello
Hello
Hello

<ul>
  <li>
    foo
  </li>
```

```
    <li>
      bar
    </li>
    <li>
      baz
    </li>
  </ul>

  nobody
  root
  daemon
```

如果你熟悉面向对象语言，可能会觉得大部分Factor代码看上去就像一遍又一遍地调用“对象”的方法。这其实是一种误解，因为任何词汇都能随意使用栈顶上的值。因此，Factor虽然还是个小语言，但是它却可以创建出可重用的、可组装的模块。

学习资源

在Factor的主页http://factorcode.org上能找到所有想要的资源。该网页包含大量关于Factor语言的文章和博文，还有很多示例代码。刷新其页面，你还可以看到不同的示例代码，这些代码都非常短小易懂。

Factor之父Slava Pestov有很多关于Factor的演讲和访谈，网络上不难找到。

最后，你能在Factor的开发环境中找到一些最新的信息，以及它自身的源代码和Factor程序。单是设置环境这一步，就能让你学到很多关于Factor的知识。

2.2.5　Fantom

Fantom是相对比较新的语言。它以前叫做Fan，不过几年前改了名字。它可以运行在JVM或者CLR之上，其目标是可以一次编写代码，然后在这两种平台上都能良好地运行，同时还修正了很多Java和C#中存在的问题。它是一种非常实用的语言，并未对语法、类库或者类型系统做大变革。它的目的只在于改善现状，创建一种可以更好地完成工作的语言。

由于Fantom必须无缝地运行在几种平台之上，因此其类库在设计时已经考虑到了将Java或者C#的特定部分抽象出来。Fantom在许多方面和Java或C#都非常相似。它是一门花括号语言，是静态类型的，但不支持范型。Fantom的创建者拒绝加入范型，是因为范型会让类型系统过于复杂。因此，他们为容器类创建了一些特定的替代方案。由于Fantom是静态类型，我们需要标明方法和字段的类型。不过，局部变量和容器的字面量有类型推演，这样用起来更容易。

Fantom有一些很酷的特性，这是我们在静态类型系统中非常需要的。对象可以接受动态调用，不过，其语法不同于常规的调用方法。这一特性再巧妙地辅助以一些元编程技巧，就能够用Fantom写出一些非常简洁而强大的程序。

Fantom的另一个特性是提倡模块化的概念。Fantom提供了不同的方式建模各个类之间的关系。既可以使用Mixin，也可以根据情况选择函数或者Actor。

在很多方面，Fantom之于Java，有点像CoffeeScript之于JavaScript。它试图消除原先语言设计中的一些缺陷，并且重新设计全新的类库，让这些类库更一致、更易用。Fantom还加入了一些Java早就该有的特性，比如Mixin和闭包。如果你熟悉Java或者C#，那么使用Fantom编程的感觉会非常自然。除此之外，Fontam还加了一些不错的特性，可以用来减少代码的行数。

和其他语言一样，我们从简单的“Hello, World”开始：

MostInterestingLanguages/fantom/hello.fan

```
class HelloWorld {
    static Void main() {
        hw := HelloWorld()
        hw.hello("Ola")
        hw.hello("Stella")
    }

    Void hello(Str name) {
        echo("hello $name")
    }
}
```

这段程序有几点需要注意。第一，和Java或者C#一样，所有东西都包在一个类中。这个类有一个`main()`方法，程序执行时会调用它。要创建`HelloWorld`的实例，只需要在类名后面加上小括号即可。Fantom其实可以有命名的构造函数，但其默认名是`make()`。如果把类名作为方法来调用，Fantom会自动调用`make()`。我创建了一个变量`hw`，`:=`语法用于告诉Fantom去推理变量的类型。然后用不同的参数调用了两次`hello()`方法，注意语句之后不需要分号。`hello()`方法接收一个参数，然后在前面加上`hello`，再将它输出到屏幕上。Fantom有一种内插字符串的简写法，即使用`$`符号。

运行上面的代码会得到如下输出：

```
$ fan hello.fan
hello Ola
hello Stella
```

如前所述，Fantom其实没有像Java或C#那样的用户自定义的范型类型。不过，Fantom支持在定义闭包和一些容器时使用范型。

MostInterestingLanguages/fantom/generic.fan

```
class Generic {
    static Void main() {
        list_of_ints := Int[1, 2, 3, 4]
        another_list := [1, 1, 2, 3, 5]
```

```
        empty_int_list := Int[,]
        empty_obj_list := [,]

        list3 := [1, 1, null, 3, 5]

        echo(Type.of(list_of_ints))
        echo(Type.of(another_list))
        echo(Type.of(empty_int_list))
        echo(Type.of(empty_obj_list))
        echo(Type.of(list3))

        map := ["one": 1, "two": 2, "three": 3]
        map2 := Int:Str[42: "answer", 26: "question"]
        empty_map := [:]
        empty_int_map := Int:Int[:]

        echo(Type.of(map))
        echo(Type.of(map2))
        echo(Type.of(empty_map))
        echo(Type.of(empty_int_map))
    }
}
```

这段代码会创建一些范型List和范型Map的例子。如果创建List或Map时没有指明元素的类型，Fantom会自己找出一个类型。在这段代码中还能看到Fantom的一个有趣的特性，即Nullable类型的概念。默认情况下，Fantom中的变量不能包含`null`。不过如果在类型名后面加一个问号，那么变量就可以是该类型的值或者`null`。默认设置为非`null`值，杜绝了很多常见的bug。这对于List和Map也是一样的。默认情况下，类型推理不会得到一个Nullable类型，但是如果用字面量创建一个容器，其中包含`null`，那么其类型就默认为是Nullable的了。

如果运行这段代码，会看到变量的类型和我们的预期是一致的：

```
$ fan generic.fan
sys::Int[]
sys::Int[]
sys::Int[]
sys::Obj?[]
sys::Int?[]
[sys::Str:sys::Int]
[sys::Int:sys::Str]
[sys::Obj:sys::Obj?]
[sys::Int:sys::Int]
```

Fantom支持轻量级的闭包与函数类型。许多情况下，使用它们和在Ruby中使用代码块基本上是一样的，唯一的区别在于，Fantom中可以更容易地给函数传递多个代码块。如果你没有为闭包定义参数列表，它会假设有一个隐式的变量，称为`it`。调用闭包的方式是在名字后面加上括号，就像调用普通的方法一样。

我们还可以在Fantom的标准类库中找到很多期望看到的东西，这些东西也能在Ruby、Groovy等动态语言中找到。

MostInterestingLanguages/fantom/closures.fan

```
class Closures {
    static Void main() {
        h := |name| { echo("Hello $name") }
        h("Ola")
        h("Stella")

        list := [42, 12, 56456, 23476]
        list2 := ["Fox", "Quux", "Bar", "Blarg", "Aardvark"]

        list.each { echo("Number $it") }

        echo(list2.sort |left, right| { right.size <=> left.size })
    }
}
```

首先，我们创建了一个闭包。它接受一个参数。然后用两个参数分别调用该闭包。接下来我们分别创建了一个整数列表和一个字符串列表。`each()`方法可以遍历一个数据容器。这里你可以看到我们如何使用隐式变量`it`，而不用定义一个参数。第二个例子接受两个参数，它会根据闭包的返回结果对列表进行排序，非常类似Java的`Comparator`。

运行代码，可以看到如下输出：

```
$ fan closures.fan
Hello Ola
Hello Stella
Number 42
Number 12
Number 56456
Number 23476
[Aardvark, Blarg, Quux, Fox, Bar]
```

虽然大多数时候Fantom都使用强静态类型，但它也可以通过动态调用这种轻量级的方式绕过类型的限制。如果你使用`->`（而不是`.`）操作符，就会产生动态调用，类型检查器将忽略它。如果调用的方法存在于目标对象上，那么它会像静态类型中那样被调用。另外，我们还可以改写一个名为`trap()`的方法，通过钩子进入进程，这样，就能决定动态调用时应该如何处理。用这种办法能够模拟一些动态语言社区中流行的非常酷的效果，同时应用的其他部分也能确保类型安全。

下面这个例子演示了如何利用方法调用和闭包生成XML：

MostInterestingLanguages/fantom/dynamic.fan

```
class XmlBuilder {
    Str content
    Int indent
    new make() { this.content = ""; this.indent = 0 }

    override Obj? trap(Str name, Obj?[]? args) {
        this.content += "${doIndent()}<$name>\n"
        this.indent += 1
        if(args != null && args.size > 0) {
            if(args[0] is Func) {
                ((Func)args[0])(this)
            } else {
                this.content += doIndent()
                this.content += args[0]
                this.content += "\n"
            }
        }
        this.indent -= 1
        this.content += "${doIndent()}</$name>\n"
        return this.content
    }
    Str doIndent() {
        Str.spaces(this.indent * 2)
    }
}
class Dynamic {
    static Void main() {
        x := XmlBuilder()
        x->html {
            x->title("ThoughtWorks Anthology")
            x->body {
                x->h1("Welcome!")
            }
        }

        echo(x.content)
    }
}
```

XmlBuilder类记录当前的内容和缩进的层次。它改写了trap()方法，并在方法内部做了很多事情。它首先打印一个开标签，将其加入到字符串content中，然后改变缩进层级。随后检查是否有参数，如果有，再检查是不是函数。如果是函数，则直接执行它。如果不是，就把参数追加到字符串content中。最后还原缩进级别，加入闭标签。

有了这一机制后，用main()方法创建XML文档就非常简单了，只要调用XmlBuilder实例上与标签名一致的方法即可。执行这段代码，将得到期望中的结果：

```
$ fan dynamic.fan
<html>
  <title>
    ThoughtWorks Anthology
  </title>
  <body>
    <h1>
      Welcome!
    </h1>
  </body>
</html>
```

Fantom是一门非常强大的语言。虽然它把自身的强大功能隐藏在花括号语法之后，不过一旦掌握了它的语法，你就会发现它能完成所有Java和C#可以完成的工作；而且实现方式更简洁、更清晰。另外，Fantom能够在不同平台之间迁移也是其优势之一！

学习资源

目前，Fantom的不足是其相关资源不多。浏览其主页http://fantom.org或许会有所收获。此外，Freenode上还有一个相关的IRC频道。

虽然从这些资源能中能小有收益，但是我初学时还是花了不少精力研究这门语言，也许你也要花费很大的时间精力。不过，这一切付出都是值得的。

2.2.6 Haskell

就本文中的所有函数式语言来说，Haskell绝对可以说是把函数式范式发挥到最为极致的语言。Haskell是一门纯函数式编程语言，也就是说不支持任何形式的可变性或副作用。当然，这条真理还是有例外的，如果Haskell不支持任何副作用，你就无法执行打印操作，也无法从用户那里获得输入。Haskell可以执行I/O，也可以实现一些看上去像副作用的操作。但是在语言的模型中，其实并没有副作用出现。

Haskell是一门惰性语言，这一特点使它本质上有别于其他语言。这也就是说，函数的参数要到真正需要的时候才会计算。这一特性简化了很多工作，比如创建无限的流、递归函数定义以及很多其他有用的事情。由于没有副作用，因此除非刻意使用这一特性，否则我们通常不会注意到Haskell是一门惰性语言。

自ML以来，函数式编程语言开始分为两大不同家族——一族使用静态类型，另一族则不使用。有些函数式语言具有很先进的静态类型系统，Haskell便是其中之一。在某些语言的类型系统中很难表达的东西，Haskell的类型系统却可以很容易地表达出来。然而，虽然类型系统非常强大，但它并不会在写程序时造成太大的干扰。大多数情况下，我们不必为函数或者变量名指明类型；因为Haskell自身可以通过类型推演找到正确的类型。

Haskell没有基于继承的类型系统，但它大量使用了范型。Haskell的类型系统中的很大一部分都是类型类（type class）。这些类让我们可以向已有的类型中添加不同的行为。我们也可以把类型类看做一个接口，只不过在它定义好之后，包含了一些添加到类中的实现。这是Haskell非常强大的特性，一旦用过这种类型类，在使用其他语言时就会非常想念它。

总的来说，Haskell是一门非常强大的语言。许多领域的研究者都能用Haskell进行进一步开发，因此，很多有趣的程序库最初都是用Haskell实现的。举例来说，Haskell支持很多不同的并发范式，其中包括软事务存储（STM）以及嵌套数据并行。

用Haskell编写“Hello World”程序，作为起始的例子可能有些怪异，这是因为在Haskell中进行I/O操作多少还是有点儿复杂。不过没关系，先来看一下程序代码：

MostInterestingLanguages/haskell/hello.hs

```
module Main where

main = do
  hello "Ola"
  hello "Stella"

hello name = putStrLn ("Hello " ++ name)
```

若想将这段程序做为单独的文件运行，必须在名为Main的模块中定义一个main()函数。do关键字让我们可以一次完成好几件事。最后，我们定义了一个hello()函数，它接收一个参数，将参数与*Hello*拼接，然后打印它。

编译并运行代码，可以得到如下输出：

```
$ ghc -o hello hello.hs
$ ./hello
Hello Ola
Hello Stella
```

与Erlang一样，Haskell也非常适合模式匹配。我还没有提到这一点，不过空格在Haskell中非常关键，也就是说，Haskell依靠空格确定代码结构，这一点和CoffeeScript与Python很像。应用模式匹配时，程序看上去会相当整洁。下面的代码创建了一个数据类型表示形状，然后使用模式匹配计算不同形状的面积。在这段代码中，我们又见到了使用递归和模式匹配实现反转队列的例子。

MostInterestingLanguages/haskell/patterns.hs

```
module Main where

type Radius = Double
type Side   = Double

data Shape =
      Point
```

```
         |  Circle    Radius
         |  Rectangle Side Side
         |  Square    Side
area Point            = 0
area (Circle r)       = pi * r * r
area (Rectangle w h) = w * h
area (Square s)       = s * s
rev [] = []
rev (x:xs) = rev xs ++ [x]

main = do
  print (area Point)
  print (area (Circle 10))
  print (area (Rectangle 20 343535))
  print (area (Square 20))
  print (rev [42, 55, 10, 20])
```

上述代码输出如下：

```
$ ghc -o patterns patterns.hs
$ ./patterns
0.0
314.1592653589793
6870700.0
400.0
[20,10,55,42]
```

在Haskell中，我们可以看到大多数函数定义看上去很像代数公式。定义像Shape这类数据类型时，需要列出所有可能性，还要列出每种可能性需要的数据。然后，当我们根据运行时的不同数据派发到不同的area()方法时，也要挑出数据类型中包含的数据。

前文提到过Haskell是惰性语言。证明这一点很容易，例如定义一些会无限执行的操作：

MostInterestingLanguages/haskell/lazy.hs

```
module Main where

from n = n : (from (n + 1))

main = do
  print (take 10 (from 20))
```

这段代码看上去相当简单。take()函数是在Haskell核心类库中定义的。take()会从给定的列表中取出指定数目的元素（本例中是10个）。函数from()使用冒号构建新的列表，该列表定义成n的值，它后面跟着一个列表，通过n+1再次调用from()产生。在大多数语言中，一旦调用这个函数就会陷入无限递归，程序也就崩溃了。但是Haskell只会有限次地调用from()，直到取得足够用的值为止。这一点非常难懂，需要花些时间才能理解。不过请记住，这里的冒号没有什么特殊之处，它不过是Haskell的求值方式而已。

运行代码后的结果如下：

```
$ ghc -o lazy lazy.hs
$ ./lazy
[20,21,22,23,24,25,26,27,28,29]
```

关于Haskell，我想展示的最后一件事是关于类型类的。由于Haskell不是面向对象的，也没有继承，因此有些事做起来非常笨重，比如定义范型函数、执行打印、检查相等性或者其他类似的工作。类型类可以解决这个问题，基本上，它允许我们根据不同的Haskell类型变换不同的实现。类型类异常强大，并且与之前所见的传统的面向对象语言截然不同。让我们看一个例子：

MostInterestingLanguages/haskell/type_classes.hs

```haskell
module Main where

type Name = String

data Platypus =
    Platypus Name
data Bird =
    Pochard Name
  | RingedTeal Name
  | WoodDuck Name

class Duck d where
  quack :: d -> IO ()
  walk  :: d -> IO ()

instance Duck Platypus where
    quack (Platypus name) = putStrLn ("QUACK from Mr Platypus " ++ name)
    walk  (Platypus _) = putStrLn "*platypus waddle*"
instance Duck Bird where
    quack (Pochard name) = putStrLn ("(quack) says " ++ name)
    quack (RingedTeal name) = putStrLn ("QUACK!! says the Ringed Teal " ++ name)
    quack (WoodDuck _) = putStrLn "silence... "
    walk _ = putStrLn "*WADDLE*"

main = do
  quack (Platypus "Arnold")
  walk (Platypus "Arnold")
  quack (Pochard "Donald")
  walk (Pochard "Donald")
  quack (WoodDuck "Pelle")
  walk (WoodDuck "Pelle")
```

这段代码包含了很多信息。首先，我们定义了两种数据类型：一种是Bird（鸟类），一种是Platypus（鸭嘴兽），它们都有一个名字（Name）。接下来我们创建一个类型类，名为Duck（鸭子）。我们知道如果某种动物叫声像鸭子，走路姿势也像鸭子，那么它就是鸭子。因此，类型类Duck定

义了两个函数，quack()和walk()。这些声明只是规定了参数的类型和返回值的类型。这些类型签名表明，它们接受一个像鸭子的东西，然后打印一些输出。随后，我们定义了类型类Platpus的一个实例。我们只在实例中定义必要的函数，这些函数和Haskell的顶层函数并无区别。之后，我们对Bird做了同样的事情。最终，我们实际上是在不同的数据实例上调用了quack()和walk()。

运行此示例，可以看到和预期相符的输出：

```
$ ghc -o type_classes type_classes.hs
$ ./type_classes
QUACK from Mr Platypus Arnold
*platypus waddle*
(quack) says Donald
*WADDLE*
silence...
*WADDLE*
```

类型类相当强大，但是在这样一小段代码中只是管中窥豹。不过请放心，一旦彻底理解了类型类，你就迈入了精通Haskell的大门。

学习资源

学习Haskell的最佳起点是一本在线图书，名为*Learn You a Haskell for Great Good*（http://learnyouahaskell.com）。该书以简单生动的方式，带你循序渐进地学习Haskell。

除此以外还有一些涉及Haskell的书，不过却大同小异，大部分都注重从数学或者计算机科学的角度讲述如何使用Haskell。如果你想学习如何将Haskell用做一种通用目的编程语言，最好读一读Bryan O'Sullivan、Don Stewart和John Goerzen合著的*Real World Haskell* [OGS08] 。这本书也可以在线访问http://book.realworldhaskell.org/read。

2.2.7 Io

在本文介绍的所有语言中，我认为Io绝对是我的最爱。它虽然很小，但是很强大。虽然它的核心模型简单且中规中矩，但却诞生出很多新奇、精彩的特性。

Io是纯面向对象语言，“纯”，即Io中所有事物都是对象，没有例外。所有我们碰到的、用到的，或是实现用到的都是对象，我们可以到达它，持有它。相比于Java、C#，Smalltalk以及很多其他面向对象语言，Io并不使用类。Io使用基于原型的对象系统取代类，其思想是根据已有的对象创建新对象。我们可以直接修改某个对象，然后将其当做新对象的基础。

传统的面向对象语言有两个不同的概念：类和对象。在更纯粹的面向对象语言中，类也算一种对象。不过这两者之间有着根本的差异，即类具有对象所没有的行为。在Io中，方法也是对象，就像其他东西一样。方法可以加到任何对象上。这种语言的编程模型与基于类的语言差别很大，因此，我们可以用截然不同的方式建模。基于原型的语言有一个优势，即它们能够很好地模拟基

于类的语言。因此，如果想继续使用基于类的模型，Io也不会限制我们。

Io是一门小巧的语言，但却支持很多功能。它支持一些不错的基于协程（coroutine）的并发特性。利用Io的Actor，构建健壮的、可扩展的并发程序非常容易。

还有一个方面，Io也做到了极致：用于表示Io代码的元素都是一等对象。这意味着我们可以在运行时创建新代码，可以修改已有的代码，还可以自查已有的代码。利用这一特性可以创建出极其强大的元编程程序。

在Io中，定义方法就像普通的赋值一样——首先，创建一个方法，然后把它赋给某个名字。第一次赋予名字时，要使用`:=`，之后就可以直接用`=`。我们的“Hello, World”例子如下：

MostInterestingLanguages/io/hello.io

```
hello := method(n,
  ("Hello " .. n) println)

hello("Ola")
hello("Stella")
```

我们先用`..`操作符拼接字符串，然后要求字符串打印自身。输出应该很容易猜得到：

```
$ io hello.io
Hello Ola
Hello Stella
```

Io有一个使用Actor和Future的协同多任务机制。只要调用Actor的`asyncSend()`方法，并传入要调用的方法的名字，任何Io对象都可以用作Actor。我们必须显式地调用`yield`，以确保所有代码都已运行。

MostInterestingLanguages/io/actors.io

```
t1 := Object clone do(
  test := method(
    for(n, 1, 5,
      n print
      yield))
)

t2 := t1 clone
t1 asyncSend(test)
t2 asyncSend(test)

10 repeat(yield)
"" println

t3 := Object clone do(
  test := method(
    "called" println
```

```
    wait(1)
    "after" println
    42))
result := t3 futureSend(test)
"we want the result now" println
result println
```

上述代码首先创建了新对象t1，它有test()方法，能够打印数字1~5，每次打印之间调用yield。然后再克隆这个对象，依次调用这两个对象的asyncSend(test)()方法。最后在主线程中调用10次yield。

第二段代码又创建了一个新的对象，它也有一个test()方法，它会先打印一些东西，然后等1秒，再打印一些东西，最后返回一个值。我们调用这个对象的futureSend(test)()，就可以将它转变为一个透明的Future。这个调用的结果不会立即计算，而是等到真正需要其值的时候，也就是最后一行，此时我们要打印它的结果。该特性和Haskell处理惰性值的方式非常类似，不过在Io中，我们必须显式地创建Future才能获得同样的效果。

运行程序，得到如下输出：

```
$ io actors.io
1122334455
we want the result now
called
after
42
```

两个Actor彼此交替地打印输出，从中可以看出Actor协作的本质来。你可能也注意到了，通过Future调用的方法的输出并未立刻被打印出来，这说明它是到最后一刻才调用的。

Io另一个强大特性是它支持反射和元编程；基本上Io中的任何东西都可以被访问和修改。Io中的代码是可以在运行时访问的，它以一种消息的形式展现。我们可以利用此特性完成很多事情，比如创建高级的宏设施。尽管下面这个例子可能没有什么实际的用途，但它确实展示了这种方法的威力：

MostInterestingLanguages/io/meta.io

```
add := method(n,
  n + 10)

add(40) println

getSlot("add") println
getSlot("add") message println
getSlot("add") message next println
getSlot("add") message next setName("-")

add(40) println
```

首先，这段代码创建了add方法，可以给参数值加10。我们调用它，确保它是可用的。然后，我们使用getSlot()访问方法对象，但不真正对它求值。打印方法对象，然后得到它的message

对象，之后再打印message对象。Message是对象链，因此，对一个message求值后，Io将根据next指针指向下一步操作。打印next指针的值，然后修改了next的这个message名字。最后，重新以40为参数调用add。基本上，这段代码就在运行时期动态地修改了add()方法的实现。

运行代码，可以看到相应的结果：

```
$ io meta.io
50

# meta.io:2
method(n,
    n + 10
)
n +(10)
+(10)
30
```

Io可塑性极强，几乎所有东西都是可访问的、可改变的。它是一门非常强大的语言，而它的强大之处却隐藏在小巧的外表之下。我第一次学习Io时就被它震撼到了，而且在之后的学习过程中，它也不断地改变着我的想法。

学习资源

目前还没有介绍Io的书。不过http://www.iolanguage.com/scm/io/docs/IoGuide.html上的入门教程是个不错的起点。看完这个教程后，你可以读一读参考文档，尝试理解Io的一些要点。同时，Io也非常易于理解，我们通常可以直接查看某个对象包含了什么功能。

Steve Dekorte做过一些关于Io的在线演讲和访谈。Bruce Tate的《七周七语言：理解多种编程范型》一书中也有关于Io的章节。

2.3 总结

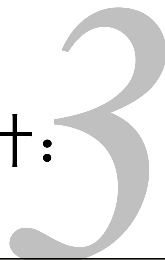
如果你对编程语言感兴趣，那么这是属于你的时代。我已经展示了一些不同的语言，以及每种语言中有趣的特性。从现在开始，就轮到你了。找寻其他有趣的语言，看看用这些语言能做些什么。学习新语言的成效在于，它会改变我们使用主流编程语言的方式。如果尝试一种编程范式完全不同的语言，那么它对你的影响将尤为明显。

我还有很多有趣的语言想在这一章里来讨论。但是，这会让这一章变成一本书。这些语言（有新有旧）包括（没有特定顺序）：Frink、Ruby、Scala、Mirah、F#、Prolog、Go以及Self。

我写本文时正值新年前夕。在这一天有一个传统，就是要为来年做些打算。对于程序员来说，选择并学习一门新语言是个传统项目——我是从《程序员修炼之道》这本书开始的，建议你也这么做。这会让你成为一名更优秀的程序员。

第 3 章

面向对象程序设计：对象优于类



Aman King撰文

就在几年前，如果你问我要一个面向对象的解决方案，我会列出所有的类，其中包含数据属性、方法签名和精心组织的继承结构，而你也会欣然接受。

今天，我向你保证，我只会给你一个可用的系统，其中包括大量能够运行的对象，我所做的一切都是为了解决你的问题。在大多数情况下，我敢肯定这个系统就是你需要的。

那么，过去和现在有何不同呢？

变化之一是我们多数人写代码的方式。以前我们都是预先设计，然后再实现代码。今天，我们在实现的过程中同时也在设计。对于测试驱动开发的执着让我们不断地改进代码，包括外部接口和内部实现。

变化之二是我们所用的程序设计语言和程序库。以前所有的程序都是Java编写的。今天，也许最先在Ruby项目上会用Rails、ActiveRecord等技术，之后却在Java项目上改用Struts 2、Hibernate，不一而足。我们也在JavaScript上投入了很多精力。

这些变化都会影响程序员对于"面向对象编程"的思考。

通过测试驱动代码，敏捷开发者学会了如何避免做一些不成熟、却影响广泛的设计决策。

在不同的语言之间跳转，就得仔细地看看常见的编程范式。Java、Ruby和JavaScript，所有的面向对象语言对于基础概念都有不同的解读，这些差异有时很细微，有时却很悬殊。

本文不会讨论测试驱动开发，不是面向对象设计的教程，也不是语言的对比。我要探讨的问题是在面向对象编程中，"对象优于类"这一观点的影响力。也许有些观点会影响到你的设计和实现策略。

3.1 对象优于类

“对象优于类”是什么意思？

好吧，这个想法来自于Java中的“对象高于类”、Ruby中的“类即对象”以及JavaScript中的“用对象取代类”。

试想某个业务问题的软件解决方案，无论何种解决方案，最终都会是某种形式的应用程序，并且需要一个实现功能的运行时环境。该环境将执行大量的指令，多种指令结合起来，使业务问题最终得以解决。

作为程序员，我们要理解这个环境，毕竟，我们是那个启动了一切的人，要说清楚将要发生什么、何时发生、如何发生。

这正是面向对象范式的用武之地。它提供了一个思维模型，帮我们预想事情在运行时环境中如何发生。我们可以将整个系统看作是一个生态系统，不同的对象在其中彼此交互。每个对象都是一个实体，告诉其他实体应该做什么，并且能响应其他实体的请求，改变自身的状态。这些实体要么保守内向，要么热衷交流，要么转顺即逝，要么寿命长久。

不过，不同的人会用不同的范式描述运行时环境，这其中包括过程式、函数式、事件驱动，等等。很明显，目前面向对象是最流行的。这种范式通过使用面向对象语言进行面向对象编程实现。这些语言提供了用以指定对象行为和状态的构造。

尽管面向对象语言的共同点要多于不同点，不过它们还是有些分别，即给程序员提供的展现模型有所不同。只有两门面向对象语言的区别很明显时，才需要回首思索。

除了对象，面向对象语言的另一个重要概念是类。这是大多数面向对象语言都提供的构造，用以为创建对象而定义一个模板或蓝图。

这些传统的“类”构造如何帮助我们呢？

在宏观层次上，它们能够展现一个关于系统的静止不变的视图，就像类之间相互连接的网络。它们可以通过继承、组合、协作的方式联系起来。

在微观层次上，类代表了独立的领域概念，承载了数据属性和操作。通常，类会在继承的体系中共享一些通用的属性，以此创建一种分类方法，模拟现实世界。

在低层次上，类定义了实例交互的契约以及实现方式。

但是，对于运行中的应用程序，类的角色是什么呢？一旦系统启动，几乎所有工作都是由对象在运行环境中完成的。由此，类成了一个被动的角色，只有在创建新实例时，才扮演引用的模板。但也有例外，即在运行时调用类级的方法。

如前所述，在应用程序的概念形成或构造阶段，类扮演了非常重要的角色。因此，我将它们看作是用来表现系统的静态表现的设计工具。而在运行时，类的作用非常有限。

Trygve Reenskaug创建了MVC模式，参与开发了UML建模语言。关于类的作用，他曾在[RWL95]中有过如下表述：

> “类/对象的二元性对面向对象编程和面向对象建模都很重要。”

认识二者的区别及其产生的影响正是我要探讨的主题。如果我们认为运行中的软件是最重要的，那么是不是可以说，运行时环境和静态表现一样重要呢？当我们进行设计和实现时，除了考虑类之外，难道不应该偶尔要重视一下对象吗？

3.2　类关注与对象关注

我们首先看一看“类关注”与“对象关注”两种方法有何不同。

在我看来，“类关注”方法的驱动力就是以业务可识别的方式对领域进行建模。除了捕捉静态快照的每个细节，它不会直接表现出领域概念是如何随着时间变化而相互影响的。典型的基于类的解决方案，是在给定的表述问题的语句中，标出所有名词，然后为每个名词创建一个类。现实中概括与具体的关系会反映到面向对象语言中的继承关系。典型的企业应用程序都会有下面几个类：`Person`、`Employee`、`Manager`、`SeniorManager`，等等。

而对于“对象关注”的方法，运行时和对象之间的关联与交互将成为设计背后的驱动力。它会导致在不同的时间点上，领域对象可以扮演不同的角色。在多个类型的领域实体之间，这些角色可能有重合。比如，企业应用程序中的角色可能包含：`Billable`、`NonBillable`、`Delegator`、`Approver`、`Reportee`，等等。

3.2.1　角色的角色

我们来深入分析一下前面的例子。在企业中，高级经理（`Senior Manager`）毫无疑问是一类特殊的经理（`Manager`），而经理也是一名员工（`Employee`），员工则很明显是人（`Person`）。然而在运行时环境中，比如说在办公室里，你见过高级经理是由不同的部分组成的吗，一部分像光彩照人的员工，一部分像个幽灵般的人？如果只停留在这些表象上未免太可笑了。事实上，高级经理是一个完整的实体，他扮演许多关键的角色。如果有一项申报等待批准，他就是那个审批者。如果需要报告工时单，他就是报告者。如果他休假，就会把这些工作委托给经理，这名经理就成为审批者和报告者。

如果忽略角色，那么上例中领域的类结构可能对应如下的Java代码：

```
class Person {
  // ...
}
class Employee extends Person {
  // ...
}
class Manager extends Employee {
  // ...
}
class SeniorManager extends Manager {
  // ...
}
class SalesAssociate extends Employee {
  // ...
}
```

然而，如果强调领域实体扮演的角色，可能会得到如下代码：

```
interface Approver {
  // ...
}
interface Delegator {
  // ...
}
interface ProposalWriter {
  // ...
}
class SeniorManager implements Approver, Delegator {
  // ...
}
class Manager implements Approver {
  // ...
}
class SalesAssociate implements ProposalWriter {
  // ...
}
```

注意观察每个角色是如何由Role Interface[①]（角色接口）展现的。所有领域实体，诸如`SeniorManager`、`Manager`等都实现了相应的角色。同样请注意新的实现完全没有使用继承。那么代码如何重用呢？很明显，高级经理和经理所执行的审批过程是有相似性的。

在运行时环境中，如果继承没有扮演一个可见的角色，那么它就退化成了单纯的代码重用或避免重复的技术。但是重用的办法有很多，比如委托和组合。因此，组合也许可以成为替代继承的一种选择[②]。我们来看看如何完成：

① http://martinfowler.com/bliki/RoleInterface.html

② http://c2.com/cgi/wiki?DelegationIsInheritance

```java
interface Approver {
  ApprovalDecision decideApproval(ApprovalRequest approvalRequest);
}
class MediumLevelApprovalStrategy implements Approver {
  public ApprovalDecision decideApproval(ApprovalRequest approvalRequest) {
    // …… 一些业务决策规则
    return approvalDecision;
  }
}
class LowLevelApprovalStrategy implements Approver {
  // ...
}
class SeniorManager implements Approver, Delegator {
  Approver approvalStrategy = new MediumLevelApprovalStrategy();
  public SeniorManager(String name) {
    // ...
  }
  public ApprovalDecision decideApproval(ApprovalRequest approvalRequest) {
    return approvalStrategy.decideApproval(approvalRequest);
  }
  // ...
}
class Manager implements Approver {
  Approver approvalStrategy = new LowLevelApprovalStrategy();
  public Manager(String name) {
    // ...
  }
  public ApprovalDecision decideApproval(ApprovalRequest approvalRequest) {
    return approvalStrategy.decideApproval(approvalRequest);
  }
  // ...
}
```

如代码所示，领域实体实例化一个其自行选择的策略，继而信任这些策略所做的决定。在本例中，经理使用低层审批策略，高级经理使用中层审批策略。这种方法非常清晰，消除了改写特定方法、模板方法、私有访问级别或保护访问级别等的复杂性，而这些正是继承结构中较为复杂的特性。

此外，这样还有一个好处，即重用发生在运行时。如果实际情况有变，改变已有的依赖关系并不难。比如，高级经理在某些特定的条件下，可以把他的审批策略临时修改为低层策略，稍后可以再改回中层策略。如果使用继承则很难做到这一点，因为实现机制在编译时已经锁定了。

再考虑另一个例子：这次的问题是要生成一个投票者列表。投票者是年满18周岁或高于18周岁的本国公民。为了增加一些复杂性，我们声明投票者也可以是国家，国家可以在所属的国家理事会（councils of nations）中进行投票。有了上面的领域实体，可以得到下面的类：

ObjectsOverClasses/java/voterlist/classbased/VoterListClassBased.java

```
class CouncilOfNations {
    private Collection<Country> memberNations;
    public CouncilOfNations(Collection<Country> memberNations) {
        this.memberNations = memberNations;
    }
    public boolean contains(Country country) {
        return memberNations.contains(country);
    }
}

class Country {
    private String name;
    public Country(String name) {
        this.name = name;
    }
}

class Person {
    private String name;
    private int age;
    private Country country;
    public Person(String name, int age, Country country) {
        // ...
    }
    public boolean canVoteIn(Country votingJurisdiction) {
        return age >= 18 && votingJurisdiction.equals(country);
    }
}

abstract class AbstractVoterList<T, X> {
    private Collection<T> candidateVoters;
    public AbstractVoterList(Collection<T> candidateVoters) {
        this.candidateVoters = candidateVoters;
    }
    public Collection<T> votersFor(X votingJurisdiction) {
        Collection<T> eligibleVoters = new HashSet<T>();
        for (T voter : candidateVoters) {
            if (canVoteIn(voter, votingJurisdiction)) {
                eligibleVoters.add(voter);
            }
        }
        return eligibleVoters;
    }
    protected abstract boolean canVoteIn(T voter, X votingJurisdiction);
}
class PersonVoterList extends AbstractVoterList<Person, Country> {
    public PersonVoterList(Collection<Person> persons) {
        super(persons);
    }
```

```
    protected boolean canVoteIn(Person person, Country country) {
        return person.canVoteIn(country);
    }
}
class CountryVoterList extends AbstractVoterList<Country, CouncilOfNations> {
    public CountryVoterList(Collection<Country> countries) {
        super(countries);
    }
    protected boolean canVoteIn(Country country,
                                CouncilOfNations councilOfNations) {
        return councilOfNations.contains(country);
    }
}
```

上面的代码可以这样调用：

ObjectsOverClasses/java/voterlist/classbased/VoterListClassBased.java

```
Country INDIA = new Country("India");
Country USA = new Country("USA");
Country UK = new Country("UK");
Collection<Person> persons = asList(
    new Person("Donald", 28, INDIA),
    new Person("Daisy", 25, USA),
    new Person("Minnie", 17, UK)
);
PersonVoterList personVoterList = new PersonVoterList(persons);
System.out.println(personVoterList.votersFor(INDIA)); // [ Donald ]
System.out.println(personVoterList.votersFor(USA));   // [ Daisy ]
Collection<Country> countries = asList(INDIA, USA, UK);
CountryVoterList countryVoterList = new CountryVoterList(countries);
CouncilOfNations councilOfNations = new CouncilOfNations(asList(
    USA, INDIA
));
System.out.println(countryVoterList.votersFor(councilOfNations));
// [ USA, India ]
```

如果将上面的代码转化为更关注对象的方式，我们就需要关注重要的交互点，而且不能再以它们“是什么”来命名，而要根据它们在“做什么”来命名。识别出交互点以后，我们可以使用“Extract Interface”、“Pull Up Method”、“Rename Method”等重构方法将代码结构修改为下面的样子：

ObjectsOverClasses/java/voterlist/rolebased/VoterListRoleBased.java

```
interface VotingJurisdiction {
    boolean covers(VotingJurisdiction votingJurisdiction);
}
interface Voter {
    boolean canVoteIn(VotingJurisdiction votingJurisdiction);
}
```

```
class CouncilOfNations implements VotingJurisdiction {
    private Collection<Country> memberNations;
    public CouncilOfNations(Collection<Country> memberNations) {
        this.memberNations = memberNations;
    }
    public boolean covers(VotingJurisdiction votingJurisdiction) {
        return this.equals(votingJurisdiction) ||
            memberNations.contains(votingJurisdiction);
    }
}
class Country implements VotingJurisdiction, Voter {
    private String name;
    public Country(String name) {
        this.name = name;
    }
    public boolean covers(VotingJurisdiction votingJurisdiction) {
        return this.equals(votingJurisdiction);
    }
    public boolean canVoteIn(VotingJurisdiction votingJurisdiction) {
        return votingJurisdiction.covers(this);
    }
}
class Person implements Voter {
    private String name;
    private int age;
    private Country country;
    public Person(String name, int age, Country country) {
        // ...
    }
    public boolean canVoteIn(VotingJurisdiction votingJurisdiction) {
        return age >= 18 && votingJurisdiction.covers(country);
    }
}
class VoterList {
    private Collection<Voter> candidateVoters;
    public VoterList(Collection<Voter> candidateVoters) {
        this.candidateVoters = candidateVoters;
    }
    public Collection<Voter> votersFor(VotingJurisdiction votingJurisdiction) {
        Collection<Voter> eligibleVoters = new HashSet<Voter>();
        for (Voter voter : candidateVoters) {
            if (voter.canVoteIn(votingJurisdiction)) {
                eligibleVoters.add(voter);
            }
        }
        return eligibleVoters;
    }
}
```

新代码的使用方式和前面的非常接近，区别在于现在`personVoterList`和`country-`

VoterList都是VoterList的实例。

第一个方案有效地利用了继承，结合了范型和模板方法设计模式。第一个方案是使用Role Interface实现的，它并不需要范型和明显的设计模式。

除此之外，两种方案之间还有区别。

在第一个方案里，PersonVoterList只能判断某个人（Person）是否能在某个国家（Country）里投票；CountryVoterList只能判断某个国家能否在某个国家理事会中投票。

在第二个方案里，VoterList能够检查任意类型的Voter实现是否能满足任何VotingJurisdiction实现。比如，personVoterList.votersFor(councilOfNations)会返回一个由不同国家[INDIA, USA]组成的固定列表[Donald, Daisy]。

后一种方法背后的关键在于，当出现对象产生交互时，真正重要的不是参数的对象类型（Person，Country或者CouncilOfNations），而是对象能否扮演调用者期待的角色（如Voter，VotingJurisdiction）。

前一个例子创建了一个高约束的系统，严格限制了对象的交互对象。后一个例子通过Role Interface，让对象可以展开更加自由的协作。每次交互，对象只要实现接口的一个最小集即可，这样就允许更多类型的对象可以满足接口从而进行交互。检验这些交互本身是否合理的任务就落在了消费者代码身上，这可以通过单元测试来完成。

有了“对象关注”的意识，我们应该更多地从Role Interface的角度思考，而不是Header Interface[①]。Header Interace只能帮助我们为类确定完整的契约，它反对实例做任何接口不允许的行为，同时也给实现加上了严格的限制，实现要么满足整个接口，要么完全不被接口接受。

Header Interface在一些项目中很流行，Martin Fowler这样描述这些项目中程序员的习惯：“要给每个类都加一个与之相伴的接口。”[②]

3.2.2　职责分离

“关注对象”产生的另一种效果，是让人更主动地思考责任的归属是类还是对象。记住，在运行时环境中，对象远比类要活跃。我们应该尽可能强化对象的功能，而不是注重引入类级别的行为。应该问这样一个问题：“一个类除了定义其实例的行为之外，真的还应该做别的吗？”

下面的代码是一个工具（utility）类，这种类几乎在每个项目里都有一大堆。这些类公然违背了类作为“对象模板”的职责，直接参与到运行时中去了。它们通常都是无状态的，包含多个

① http://martinfowler.com/bliki/HeaderInterface.html

② http://martinfowler.com/bliki/InterfaceImplementationPair.html

类级别的助手方法，这些方法可以将特定的输入值转化为期望的输出值。这样的方法似乎更为过程化，这并不是面向对象语言的风格。

随着项目的进展，工具类会逐渐膨胀，有时甚至会发展到令人恐惧的地步！从表面上无害的原生数据类型的包装类，到与关键的领域相关的业务逻辑，它们最终会包含所有类型的逻辑。要想重新控制住一个不断扩大的工具类是很困难的。因此，对于下面的代码，我们能对它的功能做何改动呢？

ObjectsOverClasses/java/utility/Utils.java

```java
public class Utils {
    // ...
    public static String capitalize(String value) {
        if (value.length() == 0) return "";
        return value.substring(0, 1).toUpperCase() + value.substring(1);
    }
    // ...
    private static String join(List<? extends Object> values,
                               String delimiter) {
        String result = "";
        // ...
        return result;
    }
}
```

我们可以把这些行为拆分到单独的类中，将它们变为实例级别的方法。核心类的实例可由新的包装类进行装饰。

ObjectsOverClasses/java/utility/Extensions.java

```java
class ExtendedString {
    private String value;
    public ExtendedString(String value) {
        this.value = value;
    }
    public String toString() {
        return value;
    }
    public ExtendedString capitalize() {
        if (value.length() == 0) return new ExtendedString("");
        return new ExtendedString(
            value.substring(0, 1).toUpperCase() + value.substring(1)
        );
    }
}
class ExtendedList<T> extends ArrayList<T> {
    public ExtendedList(List<T> list) {
        super(list);
    }
```

```
    public String join(String delimiter) {
        String result = "";
        // ...
        return result;
    }
}
```

下面两种用法的不同之处：

ObjectsOverClasses/java/utility/Utils.java

```
String name = Utils.capitalize("king"); // "King"

List<String> list = asList("hello", "world");
String joinedList = Utils.join(list, ", "); // "hello, world"
```

ObjectsOverClasses/java/utility/Extensions.java

```
ExtendedString extendedString = new ExtendedString("king");
String name = extendedString.capitalize().toString(); // "King"

ExtendedList<String> extendedList =
    new ExtendedList<String>(asList("hello", "world"));
String joinedList = extendedList.join(", "); // "hello, world"
```

注意，在第一个例子中这些功能是可以全局访问的。而在后一个例子中，只有在能够访问正确的对象时，消费者代码的扩展功能才是可见的。这种区别让人依稀记起了早年的全局变量。类级别的方法中也有类似的风险①，尤其是它们还能改变类级别状态的时候。

我第一次看到这种在运行时装饰对象的技术，是在Martin Fowler的《重构：改善既有代码的设计》[FBBO99]中。书中介绍了一个重构项，名为“Introduce Local Extension”。由于无法控制某个类，比如Java的String类，而导致无法直接为类添加行为时，这种做法就很有用。

对于整数、字符串、数组等核心数据类型来说，问题并不仅仅局限于类级别的方法。如果对象毫无约束地接受原生类型作为方法参数，或者对象本身是由原生类型构成的，那么与其相关的职责就可能就会放错地方。Java中一个经典的例子就是用Double表示金额，字符串表示邮政编码、电话号码及电子邮件地址。在多数项目中，这样的域都伴随着特定的格式化要求或验证规则。这时我们就会纠结在哪里放置这些逻辑。结果代码不是放在工具类中，就是放在了消费原生类型的对象中。而这两种方法都不完美。《重构：改善既有代码的设计》一书将这种代码风格称为Primitive Obsession，其中重构项包括Replace Data Value with Object和Replace Type Code with Class。

Java的CalendarAPI就是一个活生生的例子，它包含大量接受并返回原生数据值的方法。由于这个API非常笨重，因此Joda Time变得越来越流行，它更加整洁，也更面向对象。②

① http://c2.com/cgi/wiki?GlobalVariablesAreBad

② http://joda-time.sourceforge.net

3.2.3 测试的角度

“对象关注”的方法对测试有两方面的影响。

一方面，测试始终驱动着代码，一旦头脑中具有了运行时的场景（而不是类），那么设计会更加倾向于关注对象，而不是关注类。因为一次只需要处理一种对象间交互，因此只要搞清楚对象的角色，而不用考虑整个领域概念。这样，我们可以渐进地构建领域模型，一个角色接一个角色地进化系统。看到很多领域实体能够匹配识别出的角色，你也许会非常惊讶，因为用其他方法可能获取不到。

另一方面，如果设计是关注对象的，那么可测试性能够进一步得以提升。像Role Interface这样的技术更能够简化对象协作性的测试。我们可以使用Mock框架，但只要创建一些Stub就够了。另外，要避免类级别的方法，允许消费代码调用由依赖注入的协作者所提供的功能，而不必把实现硬编码到消费代码。这样就可以用Mock了。

现在，我们可以利用Stub对基于角色的`VoterList`进行单元测试，同时不需要`Person`和`Country`类：

ObjectsOverClasses/java/voterlist/rolebased/VoterListTest.java

```
public class VoterListTest {
    @Test
    public void shouldSelectThoseWhoCanVote() {
        Voter eligibleVoter1 = new VoterWithEligibility(true);
        Voter eligibleVoter2 = new VoterWithEligibility(true);
        Voter ineligibleVoter = new VoterWithEligibility(false);
        Collection<Voter> candidateVoters = new HashSet<Voter>(asList(
            eligibleVoter1, ineligibleVoter, eligibleVoter2
        ));
        Collection<Voter> expectedVoters = new HashSet<Voter>(asList(
            eligibleVoter1, eligibleVoter2
        ));
        VoterList voterList = new VoterList(candidateVoters);
        assertEquals(expectedVoters,
            voterList.votersFor(new AnyVotingJurisdiction()));
    }

    static class VoterWithEligibility implements Voter {
        private boolean eligibility;
        public VoterWithEligibility(boolean eligibility) {
            this.eligibility = eligibility;
        }
        public boolean canVoteIn(VotingJurisdiction votingJurisdiction) {
            return eligibility;
        }
    }
```

```
        static class AnyVotingJurisdiction implements VotingJurisdiction {
            public boolean covers(VotingJurisdiction votingJurisdiction) {
                return true;
            }
        }
    }
```

Steve Freeman和Nat Pryce等人在OOPSLA上发表的论文“Mock Roles, Not Objects”[FPMW04]中，分享了更多关于测试驱动角色的观点。

3.2.4　代码库里的线索

到目前为止，我们看过了“类关注”的例子，也了解了对应的“对象关注”的方案。你能在自己的项目中找到它们吗？下面我介绍一些线索。

我们用的框架需要继承吗？子类就是一种限制，只有通过继承才能重用；这种约束还体现在构造函数上。这种编译时给类加上的约束，刚好说明对基于对象的思考过少。可以尝试利用与框架耦合的类，把自己的代码插入到框架中。将域逻辑放在独立的类中，从而保证能够独立地修改它们。

有没有类，其继承深度处于三层或四层？每一层都会添加行为，过多层次的继承意味着对象会变得臃肿，但也能做很多事情。有多少继承带来的行为是和对象的使用直接相关的呢？而且大对象都和父类实现紧密相连，也很难单独地进行单元测试。如果为了实现测试桩重写很多方法，那么这种“Test-specific”的子类[①]将会非常笨重。

你是不是要关注大量的类，而且这些类之间差异很小？类数量膨胀说明你过于关注基于类的思考方式。过深的继承体系会导致大量的类。如果存在并行继承体系，这个问题会更严重。举例来说，每个`Car`的子类，比如`Sedan`和`hatchback`，都需要一个底盘（`Chassis`），于是就有了`SedanChassis`和`HatchbackChassis`。这时我们需要找到一些用于不同类的通用角色，并且用组合替代继承。

类是不是太少，而且每个都很臃肿？系统中有多个对象彼此交互，这个系统才是健康的，而让少数几个沉默的对象完成所有任务则不然。太少的类说明在运行时阶段，职责只被划分到少数几个类上了。更严重的是，如果类级别的方法非常普遍，类就包揽了本应由对象完成的任务。注意两件事之间的不同：一种是拥有独立的小类型，其对象拥有单一的职责；另一种是拥有大量骗人的小类型，通过继承共享行为，彼此的差异非常小。前者是合理的，它代表着松耦合；而后者却代表了高耦合。

① http://xunitpatterns.com/Test-Specific%20Subclass.html

类似的线索还有很多，但本文列出来的这些线索是不错的入口。此外，再配合定期的代码自查，就能明显提高代码库的质量。我所在的团队就把它当作一条实践。

3.3 “对象关注”的语言

目前为止，我讨论了对象化思考产生的影响，介绍了“对象优于类”的重要性。尽管示例使用的都是Java，但这一思想与语言无关。现在我们要转向另一类面向对象语言，它们对对象采取了不同的态度。与Java、C#等传统的语言不同，它们支持“对象关注”的设计和实现，并且很自然，也符合语言习惯。因此，当你想要发挥这些语言的优势时，可以更容易地以对象来思考问题，而不必为此绞尽脑汁。

我会首先展示Ruby和JavaScript，虽然二者都已有15多年的历史，但最近又重新引起了广泛关注。最后，我会简单介绍一下Groovy和Scala这些相对新潮的语言。

3.3.1 Ruby

Ruby把“对象是核心角色”的思想发挥到了极致。它最强大的特性，就是所有东西都是对象。[①]它甚至把类都看作对象。

ObjectsOverClasses/ruby/objects.rb

```
class Greeter
  def hi
    'hi'
  end
end
puts Greeter.new.kind_of? Object            #=> true, 实例是对象
puts Greeter.kind_of? Object                #=> true, 类是对象
puts Greeter.new.method(:hi).kind_of? Object #=> true, 方法是对象
puts proc { puts 'hello' }.kind_of? Object  #=> true, 代码块是对象
puts 1.kind_of? Object                      #=> true, 核心数据类型是对象
puts 'a'.kind_of? Object                    #=> true, 核心数据类型是对象
puts :some_symbol.kind_of? Object           #=> true, 核心数据类型是对象
puts [1,2,3].kind_of? Object                #=> true, 核心数据类型是对象
puts ({:a => 'a'}).kind_of? Object          #=> true, 核心数据类型是对象
```

再看另一种定义`Greeter`类的语法：

```
Greeter = Class.new do
  def hi
    'hi'
  end
end
```

① 代码块只有在需要的时候才会转为对象。

这段代码强调Greeter是一个全局常量，引用了Class类的一个实例。通过调用Class的new()方法可以创建它，调用的时候传递一个代码块作为参数，表示类的定义。Greater类对象本身包含一些有用的方法，如下所示。

- new()——创建类的实例，并按照要求进行初始化。
- methods()——返回所有类对象自身可以调用的方法。
- instance_methods()——返回所有类实例可以调用的方法。
- ancestors()——返回类继承体系中的所有成员，从该类开始，然后沿着继承链向上寻找。
- class_eval()——以代码块或者字符串作为参数，然后在类的上下文环境中将它们当作Ruby代码执行。该方法是Ruby元编程的核心。

Ruby把类看作是行为的容器，而不是固定的模板或者强数据类型。当类变成对象或者容器后，可以很自然地在运行时操作它，而且可以添加更多的行为。通过下面的代码，你会发现做这些事情非常容易！注意，Greeter并没有直接定义hello()和goodbye()的实例方法，它们是动态定义的。

ObjectsOverClasses/ruby/metaprogramming.rb

```
['hello', 'goodbye'].each do |greeting|
  Greeter.class_eval <<-MULTILINE_STRING
    def #{greeting}(name)
      "#{greeting} \#{name}!"
    end
  MULTILINE_STRING
end
greeter = Greeter.new
puts greeter.hello('Aman') #=> hello Aman!
puts greeter.goodbye('King') #=> goodbye King!
```

除了元编程，还有很多方法都可以向已有的类中添加行为。Ruby的类是*开放*的，也就是说类不是定义结束后就封闭了，也不是一次性的规约；它可以在运行时期间修改。Ruby的核心类也是如此。

ObjectsOverClasses/ruby/extensions.rb

```
class String # reopening core String class
  def custom_capitalize
    "#{self[0,1].upcase}#{self[1..-1]}"
  end
end
class Array # reopening core Array class
  def custom_join(delimiter)
    # ...
  end
```

```
end
puts "king".custom_capitalize #=> King
puts ['hello', 'world'].custom_join(', ') #=> hello, world
```

Ruby运行时保证类的任何实例都可以调用所有类实例的方法，而无需考虑这些方法是何时定义、如何定义的。实现这一点的原理是当一个对象调用某个方法时，它被看作是一条消息，包含名字和参数列表。对于每个调用，Ruby都会动态地将消息传递给对象查找链上的所有参与者，由底向上，直到某个参与者接收并处理这个消息。如果消息没有得到响应，就会调用对象上的一个名为`method_missing()`的特殊方法，并把消息的信息传递给该方法。默认情况下，这个方法会抛出一个`NoMethodError`错误信息，不过如果重写这个方法，会得到一些有趣的效果。

一个对象的查找链上可以包含以下参与者。

- Class——这是参与到对象继承体系中的某个类对象。这些对象的顺序和继承的顺序一致，即一个对象处在其父类和子类之间。最底层的类对象是目标对象最直接的类，顶层的类是`BasicObject`。[1]
- Module——在Ruby中，一个模块表示一个Mixin[2]。本质上它就是一堆行为的集合，可以包含于一个类中，但不必显式地继承它。在查找链中，它置于引用它的类之上。
- Eigen class——它是一个匿名隐藏类，只包含针对对象自身的一些实例方法。这些实例方法（或称为单体方法）只在特定的对象上是可调用的。Eigen class只在需要时才创建，并且将它置于查找链的最底端，甚至在目标对象实际的类之下。

下面的代码演示了前面提到的单体方法，这是另一种语言特性，它将对象视作“一等公民”；一个对象也可以包含独一无二的方法！

ObjectsOverClasses/ruby/singleton_methods.rb

```
class Person
  # 空类
end
peter = Person.new
def peter.crawl_walls # define singleton method on peter
  "Look, Ma! I can crawl walls!"
end
puts peter.crawl_walls #=> Look, Ma! I can crawl walls!
aman = Person.new
puts aman.crawl_walls # NoMethodError: undefined method 'crawl_walls'
```

如果类支持元编程，可以多次开闭，支持Mixin，或者对象可以有单体方法；这些就足以说明Ruby中契约的定义是非常松散的。这个结论同样适用于方法参数：它们没有任何数据类型的限

① Ruby1.9以前，`Object`是所有类的父类。

② http://c2.com/cgi/wiki?MixIn

制。任何对象都可以作为参数传入。一般情况下，方法只要假设传入的参数满足必要的契约即可。也就是说，查找一个特殊的契约（名为respone_to?）是否可用。

ObjectsOverClasses/ruby/duck_typing.rb

```
class Spider
  def crawl_walls
    "crawling..."
  end
end
class Person
  # empty class
end
def make_crawl(obj) # obj没有类型约束
  if obj.respond_to? :crawl_walls
    puts obj.crawl_walls
  else
    puts "cannot crawl walls"
  end
end

peter = Person.new
def peter.crawl_walls
  "Look, Ma! I can crawl walls!"
end
make_crawl(Spider.new) #=> crawling...
make_crawl(Person.new) #=> cannot crawl walls
make_crawl(peter) #=> Look, Ma! I can crawl walls!
```

这种动态类型被称为鸭子类型（duck typing）：

> “我看见一只鸟走路像鸭子，游泳像鸭子，叫声像鸭子，我就把这只鸟称为鸭子。”[①]

我们前面讨论过Role Interface，它更关心手头对象间的协作，而不是给对象定义一个完整的契约。鸭子类型使角色更加隐式。有些人会对类型安全有所顾虑。但如果单元测试覆盖率很高，就能够保证只发生合法的交互，这一点可以由测试驱动开发来保证。不过这仍有风险，好在Ruby爱好者通常为了更大的自由接受了这样的交易。作为一个参加了多个Ruby项目的人，我向你保证明这笔交易非常划算！

现在我们看一下如何在自己的项目上使用Ruby的语言特性。下述例子来自于我所参与过的项目中用到的常见模式。

程序员经常要处理数据传输对象。这类对象的典型特征，是需要基于字段的等价性检查以及访问方法。在某个项目里，我们有下面这样的类，它由XML订阅列表组成：

---

① 这段引自James Whitcomb Riley，参见http://en.wikipedia.org/wiki/Duck_typing。

ObjectsOverClasses/ruby/blog_example.rb

```ruby
class BlogPost
  attr_reader :title, :updated_at, :content
  def initialize(params)
    @title = params[:title]
    @updated_at = params[:updated_at]
    @content = params[:content]
  end
  def current_month?
    today = Time.now
    @updated_at.year == today.year && @updated_at.month == today.month
  end
  def truncated_content(word_limit = 100)
    # ...
  end
  def eql?(other)
    return true if equal?(other)
    @title == other.title && @updated_at == other.updated_at &&
      @content == other.content
  end
  alias_method :==, :eql?
  def hash
    @title.hash + @updated_at.hash + @content.hash
  end
end

class BlogAuthor
  attr_reader :name, :email
  def initialize(params)
    @name = params[:name]
    @email = params[:email]
  end
  def anonymous?
    @name.empty? && @email.empty?
  end
 def display
   return 'Unknown' if anonymous?
   @name || @email
 end
 def eql?(other)
   return true if equal?(other)
   @name == other.name && @email == other.email
 end
 alias_method :==, :eql?
 def hash
   @name.hash + @email.hash
 end
end
```

乍看上去，这些类彼此不同，并且没有重复。但是，如果把代码想象成可以在运行时操作的事物，便不难发现重构的机会。找出那些没有真正业务含义的方法，也就是那些构造函数、getter方法、散列值生成器、等价性判断，它们在两个类中是相似的，只是所用的属性不同而已。

如果遇到了这样的代码，尽管放手修改，最后应该写出下述代码：

ObjectsOverClasses/ruby/blog_example_cleanup.rb

```
class BlogPost
  include AttributeDriven
  attributes :title, :updated_at, :content # 生成的样板代码

  def current_month?
    today = Time.now
    @updated_at.year == today.year && @updated_at.month == today.month
  end
  def truncated_content(word_limit = 100)
    # ...
  end
end

class BlogAuthor
  include AttributeDriven
  attributes :name, :email # 生成的样板代码

  def anonymous?
    @name.empty? && @email.empty?
  end
  def display
    return 'Unknown' if anonymous?
    @name || @email
  end
end
```

写出这样的代码，其中还包括`AttributeDriven`所实现的小魔法，其实并不困难。背后无非是一些我们常常见到的元编程。不过还是感受一下它的威力吧！

将关键的领域职责分离出去后，剩下的代码只是一些模板了。模板能够用来生成更多的代码。除了可读性高之外，模板还带来了另一个好处，那就是创建类似的类时所花的时间和精力都更少了。这种方法也没有牺牲可测试性和可追踪性，因为可以对每一部分测试它究竟做了什么：领域类实现了业务逻辑；元编程的部分实现了代码生成。Ruby语言和框架很重视这些问题，因此提供了直接支持的技术。例如，我们可以不必自己实现`AttributeDriven`，而只需使用Ruby内置的`Struct`类即可。[1]

① http://www.ruby-doc.org/core/classes/Struct.html

修改某个类并不总是为了提高可读性或者移除模板代码。有时也许出于修复bug或者扩展第三方库的需要，也必须修改类。前面我们看到了，如果无法控制第三方的类库，可以使用Local Extension实现扩展，比如Java的String。不过在Ruby中，只要能在运行时访问一个类，它背后的类对象就可以直接拿来动态扩展功能。

在一个基于Ruby的Web项目中，我们使用Cucumber、Capybara和Selenium编写自动化功能测试。过了一段时间后，测试的运行时间增加到了无法接受的地步。我们决定并行运行单独的测试，以降低时间。这样做效果非常好，但是由于并行的进程要使用并行的浏览器实例，Selenium和浏览器通信时就变得不太稳定，如果不能及时创建连接，会导致测试失败。

不过，修复这个问题很简单：让Selenium在失败前重试几次。更有趣的是，我们在现有的类库中做了一个猴子补丁（Monkey patch），就实现了这个功能：

```
require 'retry-this'

module CapybaraParallelizationFix
  def self.included(base)
    base.class_eval {alias_method_chain :visit, :retry} # 元编程
  end

  def visit_with_retry(url)
    RetryThis.retry_this(:times => 2) do # 如果发生错误，重新尝试
      visit_without_retry url # 在浏览器里打开url
    end
  end
end
# 猴子补丁，修复Selenium驱动器
Capybara::Driver::Selenium.send :include, CapybaraParallelizationFix
```

我希望前面的例子能够说明在Ruby中的编程方式。大部分示例内容对于使用过Ruby on Rails的开发者来说并不陌生。Ruby on Rails是一个流行的Web框架，它鼓励可读性和流畅API。它的一些实现值得参考。[①]如果理解了Ruby如何处理了对象和类，就可以在每天的编程工作中实现同样整洁的API。

3.3.2 JavaScript

JavaScript也是面向对象语言，不过它更加有趣——很大程度上是因为它压根就没有类的概念！它遵循的范式为“对象取代类”。

在JavaScript中，不需要类就能创建对象。一个对象就是若干属性的集合。属性可以是数字、字符串、其他对象，甚至还可以是一个函数。JavaScript并不区分对象的数据字段和对象的方法。

① http://github.com/rails/rails

ObjectsOverClasses/javascript/properties.js

```
var donald = { name: 'Donald', age: 28,
    canVote: function() { return this.age >= 18; } };
for (property in donald) { // 在'name','age','canVote'上迭代
    console.log(donald[property]); // 类似于donale. name, donald. age,等等
}
console.log(donald.name); // Donald
console.log(donald.age); // 28
console.log(donald.canVote); // 函数引用
console.log(donald.canVote()); // Ture,调用函数返回的结果
```

构造函数可以初始化对象。这类函数本身并没有特殊之处。但如果调用时加上一个关键字`new`，JavaScript就会在函数内部设置一个特殊的`this`指针，指向新创建的空对象。任何`this`的属性都会设置到新的对象上。

ObjectsOverClasses/javascript/constructorFunction.js

```
function Person(name, age) { // 根据约定大写函数名
    this.name = name;
    this.age = age;
}

var daisy = new Person('Daisy', 25);
console.log(daisy.name); // Daisy
console.log(daisy.age); // 25
console.log(daisy.constructor == Person); // true
```

JavaScript中每个函数都是对象。它有一个属性原型，默认情况下指向一个空的对象。自定义的属性可以绑定在原型上。由构造函数创建出来的对象将会有一个到函数原型的链接。要访问这些对象属性时，首先要在对象自身上查找，如果没有，则会在原型上查找。通过这个特性，可以在所有由特定的构造函数创建的对象之间共享共有的属性。这一特性的优势在于，已创建的对象可以访问其创建之后才加入到原型中的属性。

ObjectsOverClasses/javascript/prototypeBasedProgramming.js

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
Person.prototype.sayHello = function() {
    return this.name + ' says, "Hello!"';
};
var daisy = new Person('Daisy', 25);
console.log(daisy.sayHello()); // Daisy说："Hello!"
Person.prototype.sayHelloTo = function(another) {
    return this.name + ' says, "Hello, ' + another.name + '!"';
};
var donald = new Person('Donald', 28);
console.log(donald.sayHelloTo(daisy)); // Donald说："Hello, Daisy!"
console.log(daisy.sayHelloTo(donald)); // Daisy说："Hello, Donald!"
```

基于原型的编程就是：行为通过一个公共对象定义，其他对象创建时会引用这个公共对象。

对于`String()`和`Array()`等内置的构造函数，JavaScript也会使用该方法，这样我们也可以按照自身需要给内置函数添加功能。

ObjectsOverClasses/javascript/extensions.js

```
String.prototype.capitalize = function() {
    if (this.length == 0) return "";
    return this[0].toUpperCase() + this.substring(1);
};
Array.prototype.customJoin = function(delimiter) {
    var result = '';
    // ...
    return result;
};
console.log("king".capitalize()); // King
console.log(["hello", "world"].customJoin(", ")); // hello, world
```


在一些流行的JavaScript库和框架中，能看到大量不同语言特性的应用。Prototype和jQuery就是两个很好的例子。它们给程序员提供了简单的API，却没有引入复杂的抽象层或者继承。程序员也能够以与框架一致的风格加入自己的扩展。

通过下面的例子，我们展示了jQuery如何读取和更新DOM对象的CSS属性：

```
<html>
<head>
    <script src="http://code.jquery.com/jquery-1.7.2.js"></script>
    <style>
        div { width:50px; }
    </style>
</head>
<body>
    <div id="content" style="height:100px;">some text</div>
    <script>
        jQuery(function() {
            var contentDiv = jQuery("#content");
            var borderWidth = (parseInt(contentDiv.css("width")) +
                    parseInt(contentDiv.css("height"))) / 10;
            contentDiv.css("border-width", borderWidth)
                    .css("border-style", "groove")
                    .css("background-color", "yellow");
        });
    </script>
</body>
</html>
```

jQuery基于原型的编程模式提供了整洁的API，比如`jQuery("#content").css("border-width", 15).css("border-style", "groove")`。所有jQuery对象都共享一个

通用的原型对象，该对象被jQuery.fn引用。库的大部分功能都是由这个对象提供的。所有程序员都能够使用这个对象。这样，无论是核心API，还是自定义的方法，都可以通过一致的方式和jQuery对象交互。

下面是一个自定义的集合扩展方法max()：

ObjectsOverClasses/javascript/max.js

```
var personsArray = [ { name: 'Donald', age: 28 }, { name: 'Daisy', age: 25 },
        { name: 'Minnie', age: 17 } ]; // 核心JS
var persons = jQuery(personsArray); // jQuery封装器
persons.each(function(index, person) { // jQuery的each
    console.log(person.name);
}); // 打印Donald, Daisy, Minnie

jQuery.fn.max = function(customMaxOn) { // 定制max方法
    var defaultMaxOn = (function(element) { return element; });
    var maxOn = customMaxOn || defaultMaxOn;
    var max;
    jQuery(this).each(function(index, element) {
        if (!max || maxOn(max) <= maxOn(element)) {
            max = element;
        }
    });
    return max;
};
console.log( persons.max(function(person) { return person.name; }) );
// { name: 'Minnie', age: 17 }
console.log( persons.max(function(person) { return person.age; }) );
// { name: 'Donald', age: 28 }
```

无论是从学术意义还是实践价值上来说，JavaScript的编程模型都是值得学习的。而且当今大量站点都高度依赖JavaScript和Ext JS这类富客户端的类库。JavaScript还可以从前台走向后台，比如Node.js就可以用于服务端的网络编程。

3.3.3　Groovy

Groovy是运行在Java虚拟机上的面向对象语言。它和Ruby有很多共同点，而且很多特性都是一样的，比如鸭子类型。Groovy某些特有的语言特性使它的对象非常实用，尤其是需要与Java集成时更为明显。

Groovy中的metaClass是指向对象的一个接口。Groovy运行时用该接口处理属性和方法的访问。一般的Groovy对象则利用它实现元编程。ExpandoMetaClass是metaClass的实现，它能直接操作方法和属性。下面的例子来自于Groovy文档[1]：

---

① http://groovy.codehaus.org/api/groovy/lang/ExpandoMetaClass.html

ObjectsOverClasses/groovy/metaClass.groovy

```
class Student {
    List schedule = []
    def addLecture(String lecture) { schedule << lecture }
}
class Worker {
    List schedule = []
    def addMeeting(String meeting) { schedule << meeting }
}
def collegeStudent = new Object()
collegeStudent.metaClass {
    mixin Student, Worker
    getSchedule {
        mixedIn[Student].schedule + mixedIn[Worker].schedule
    }
}
collegeStudent.with {
    addMeeting('Performance review with Boss')
    addLecture('Learn about Groovy Mixins')
    println schedule
    // [Learn about Groovy Mixins, Performance review with Boss]
}
```


你会看到`collegeStudent`获得了`Student`和`Worker`的行为，因为这些行为都混合到了对象的`metaClass`中。

Java的接口也能在Groovy中实现。就像鸭子类型一样，任何对象都能成为Java接口的实现。如果对象没有实现接口的方法，调用这个方法时会导致运行时异常，不过Java仍然接受该对象作为特定接口的实现。下面例子中的Groovy代码实现了Java的`Iterator`和`Transformer`接口[①]。

ObjectsOverClasses/groovy/interfaceImplementations.groovy

```
import org.apache.commons.collections.*

impl = [
    i: 10,
    hasNext: { impl.i > 0 },
    next: { impl.i-- }
]
iterator = impl as Iterator
toSquare = [ transform: {e -> e * e} ] as Transformer
println CollectionUtils.collect(iterator, toSquare)
// [100, 81, 64, 49, 36, 25, 16, 9, 4, 1]
```

① http://groovy.codehaus.org/Groovy+way+to+implement+interfaces

3.3.4　Scala

Scala是一门运行在Java虚拟机上的静态类型语言。《Scala编程》[OSV08]一书中提到，Scala拥有一个先进的静态类型系统，而且还有类型推演机制。虽然Scala关注于类型，不过有些特性还是给了对象足够的重视。对于初学者来说，Scala中的每个值都是对象，运算符则是对象方法的调用。

也许更为关键的是Scala没有`static`这个关键字：也就是说该语言不接受类级别的成员。这样就杜绝了一个类除了定义实例样本之外，还要担负其他职责的问题。作为`static`的替代品，程序员可以使用Scala的单体对象（singleton object）。单体对象由Scala自动实例化，遵循单体模式，而且只有唯一一个实例。一个单体对象和一个类可以使用相同的名字共存。

ObjectsOverClasses/scala/singleton-objects.scala

```
class Person(val id:Int, val name:String, val age:Int) {
  def canVote() = { age >= 18 }
}
object Person { // singleton对象
  private val persons = List(new Person(1, "Donald", 28),
    new Person(2, "Daisy", 25), new Person(3, "Minnie", 17))
  def findById(id:Int):Person = {
    persons.find(person => person.id == id).get
  }
  def findAllByName(name:String):List[Person] = {
    persons.filter(person => person.name == name)
  }
}
var person = Person.findById(2)
println(person.name + " can vote: " + person.canVote())
// Daisy can vote: true
```

我希望本文能告诉大家，面对同一种面向对象的编程范型，不同的语言会有风格迥异的处理方式。就我个人而言，每种方式都魅力十足！

3.4　要点回顾

本章即将结束，下面快速回顾一下已经讨论过的要点。

- 面向对象范型是描述运行时系统并进一步理解、控制它的一种方式。
- 由该范型出发，运行时系统应该充满了欢快的工作对象，十分灵活，为我们解决问题！
- 在运行时，类通常都没有什么积极的作用，但它在系统的设计和构建阶段最为有用。
- 认识到对象和类之间的不同点非常重要，这会影响我们的设计和实现决策。
- 使用关注对象的方法构造系统，而不是关注类的方法，这样就可以得到一个灵活、开放的系统，不同对象也能够自由地交流。

- 从现在开始：放弃继承，尤其是那些复杂的关系；记住，对象扮演的角色比现有角色的分类更有趣。
- 测试驱动开发将引导我们进行关注对象的设计。
- 有些语言的特性将对象视为“一等公民”。Ruby则将类看作对象，可以在运行时操作。而JavaScript根本就没有类的概念，只处理对象。
- 研究一下支持Mixin、元编程、原型等特性的语言。这些语言能让代码库更整洁，团队生产力更高，而且这些技术不是框架作者的专利。

3.5 总结

最后一个问题：“在对象和类的较量中，我们应该站在哪一边？”

这个问题没有绝对的答案，我们必须自己来判断。不过我这里总结一下自己对这些概念的理解。

什么是对象？对象是有“生命”的小家伙，能表现出某些行为，和其他对象打交道，最后“死掉”或者只是被遗忘，而它所做的一切都是为了解决问题。

什么是类？类是个容器，容纳了相关行为，新的对象从此起步。

那么哪些不是类？类不是面向对象系统的基础构件，对象才是！类既不是强制的契约，也不应该限制对象的行为。

我们为什么需要类呢？

- 类能改进代码的结构，提高可读性，因此提升了系统的可维护性。
- 类可以把一组相关的行为组织到一起，并且赋予其一个有领域含义的名称，因此也是和团队成员、业务相关人员进行沟通的工具。
- 类有助于设想和理解系统的静止快照；如果只有对象的话，系统将是高度动态、不断变化的。

上述内容都是类带来的“软效益”。其他实在的好处在一些语言里不需要类也能实现。无论哪种方式，软效益都扮演了指导原则的角色。一个简单的指标可以判断我们是不是正确地使用了类：与类相关的设计决策是不是让我们感到绊手绊脚？系统中的对象是不是可以不受约束地移动，自由自在地与需要的对象进行交互？

你知道吗？你我都爱无拘无束，因此我们一定要继续探寻激动人心的程序世界，找到下一个给予我们挑战的语言，激发我们不断的思考的动力！

第4章 使用面向对象语言进行函数式编程

Marc Needham撰文

近几年来，函数式编程语言的普及程度有所提升，一些函数式编程的技巧也颇受欢迎。即使我们正在使用的是一门以面向对象为其主要特征的语言，我们也会从中受益匪浅。

虽然只是在最近几年，函数式编程的理念才开始受到越来越多的关注，但早在2005年左右，CLR就已经提供了底层的平台特性，让我们可以用C#去写函数式的代码了。

之后，C#也不断演进，以至于现在我们已经可以用C#写出看起来和F#颇为相似的代码。F#是微软推出的一门函数式编程语言，最近Visual Studio已经为F#提供了“一等公民”的支持。

函数式编程的理念[1]至少发端于半个世纪之前。

在本章中，我们将展示一些函数式编程技巧的实例。尽管这些实例是用C#和Ruby编写的，但是其理念同样也适用于其他类似的语言，比如Scala或Java。

4.1 集合

在试着理解如何用函数式编程解决问题之前，我们先来考虑一下自己看待集合的方式。

4.1.1 转换思维

学习函数式编程时，最有趣的思维方式转变是集合的处理方式。

采用命令式的方式会孤立地看待集合中的每一个元素，通常我们会用`for each`循环遍历集合。

① http://en.wikipedia.org/wiki/Functional_programming

如果用函数式编程处理与集合相关的问题，我们就该把集合看做一个整体，Patrick Logan将其称之为“转换思维[①]”。

在考虑该用哪些函数来对集合做处理之前，我们会先观察集合的初始状态，并描绘集合转换后的最终状态：

初始状态→ () → () → () → 最终状态

这和管道过滤器式的架构非常相似。在管道过滤器中，数据流经一个管道，处理数据的函数则在管道中充当过滤器。

这种处理集合的方式，可以通过Bill Six所称的“函数式集合模式[②]”方式实现。

对集合的操作主要可以分为三种类型。

1. 映射

映射模式将一个函数应用于集合中的每个元素，并返回一个新的集合，其中包含每个函数应用的结果（见图4-1）。这样，如果要获取一组人的名（first name），可以用如下代码：

```
var names = people.Select(person => person.FirstName)
```

替代如下的命令式代码：

```
var names = new List<string>();
foreach(var person : people)
{
   names.Add(person.FirstName);
}
```

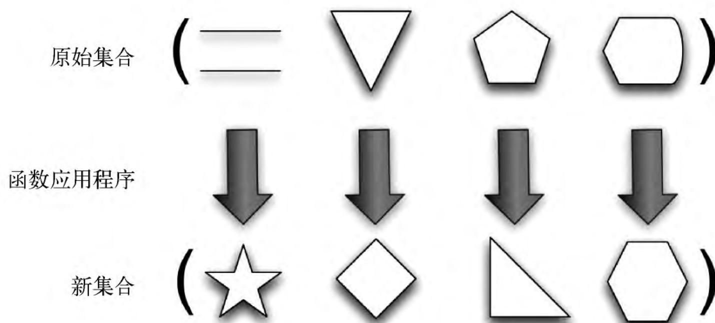
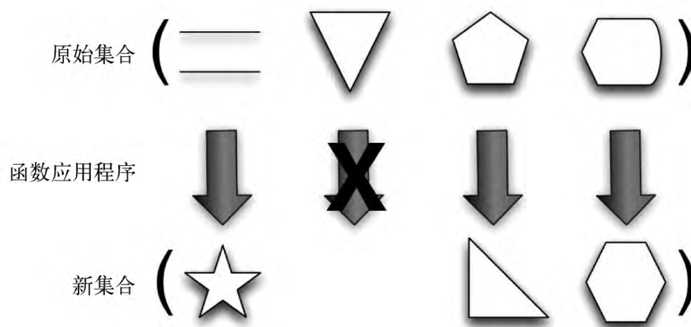


图4-1　映射函数

① http://www.markhneedham.com/blog/2010/01/20/functional-collectional-parameters-some-thoughts/#comment-30627
② http://www.ugrad.cs.jhu.edu/~wsix/collections.pdf

2. 过滤

过滤模式将谓词函数应用于集合中的每个元素，并返回一个新的集合，其中包含了谓词函数返回`true`的元素。



如果要获取年龄超过21岁的人，可以写出如下代码：

```
var peopleOlderThan21 = people.Where(person => person.Age > 21);
```

而下述命令式代码虽与上述代码等价，但可读性较低：

```
var peopleOlderThan21 = new List<Person>();
foreach(var person : people)
{
   if(person.Age > 21)
   {
      peopleOlderThan21.Add(person);
   }
}
```

3. 归约

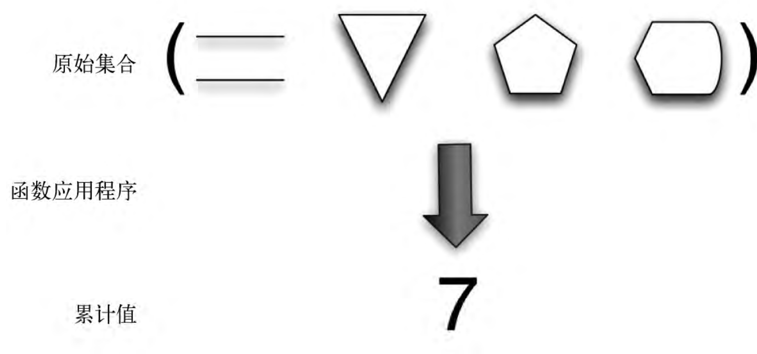
归约模式通过一个用户提供的函数，逐个将集合中的每个元素组合起来，继而将一个集合转换成一个值。

如果要得到一组人年龄的总和，可以写出如下代码：

```
var sumOfAges = people.Aggregate(0, (sum, person) => sum + person.Age);
```

下面是等价的命令式代码：

```
var sumOfAges = 0
foreach(var person : people)
{
   sumOfAges += person.Age;
}
```



4.1.2 拥抱集合

习惯了用函数式的方式操作集合之后，会发现很多问题都可以用集合解决，而此前我们则是用其他方式解决的。

很多时候，我都发现函数式的解决方案能够更加准确地描述要解决的问题。

举个简单的例子，要获得一个人的全名，可以写出如下代码：

```
public class Person
{
    public string FullName()
    {
        return firstName + " " + middleName + " " + lastName;
    }

}
```

这段代码的功能没有问题，但是，同样功能也可以写成下面这样：

```
public class Person
{
    public string FullName()
    {
        return String.Join(" ", new[] { firstName, middleName, lastName });
    }
}
```

这个例子中，第二段代码的改动并不明显，因为第一段代码中本就没有太多的重复。然而，如果要操作的数据数量增多，那么用集合来解决问题就显得很有优势了。

一个很常见的问题是比较两个值的大小，并返回较小的值。下面是一种很典型的解法：

```
public class PriceCalculator
{
    public double GetLowestPrice(double originalPrice, double salePrice)
    {
        var discountedPrice = ApplyDiscountTo(originalPrice);
        return salePrice > discountedPrice ? discountedPrice : salePrice;
    }
}
```

但也可以通过下述代码实现：

```
public class PriceCalculator
{
    public double GetLowestPrice(double originalPrice, double salePrice)
    {
        var discountedPrice = ApplyDiscountTo(originalPrice);
        return new [] { discountedPrice, salePrice }.Min();
    }
}
```

第二种解法更易于理解，因为这段代码读起来，就像在说“返回discountedPrice和salePrice中较小的那个”，而这句话刚好完美地描述了我们想要做的事情。

4.1.3　勿忘封装

LINQ库的面世让操作集合变得更简单，但很不幸的是，它也带来了副作用——现在我们更倾向于把集合作为参数传来传去。

把集合作为参数传递本身并不是问题所在，问题在于这会导致很多重复的集合操作代码，而这些代码本可以封装在定义集合的类里。此外，这样带来的另一个问题是，集合可能会在声明的类之外随意修改。

我曾经参与的一个项目，越到项目后期，就越是搞不懂一些稀奇古怪的元素是如何进入集合的，因为代码中的很多地方都可以向集合中添加或删除元素。

在给领域概念建模时，会需要很多集合操作。而多数情况下，C#的集合API并不能提供所需的全部集合操作。我们通常会怪罪到LINQ头上，但问题的根源更可能是，我们把集合用在了错误的地方。

下面就是一个典型的传递集合的例子：

```
company.Employees.Select(employee => employee.Salary).Sum()
```

我们很可能会在不同的地方重复这段代码，而如果这段计算的逻辑再复杂一些的话，麻烦就更大了。

这段代码其实可以放到Company类里：

```
public class Company
{
    private List<Employee> employees;
    public int TotalSalary
    {
        get
        {
            return employees.Select(employee => employee.Salary).Sum();
        }
    }
}
```

有时，我们需要更进一步，给集合创建一个包装类。

比如，可能会有一个Division类，它需要向外暴露该部门全体员工的总工资：TotalSalary：

```
public class Division
{
    private List<Employee> employees;
    public int TotalSalary
    {
        get
        {
            return employees.Select(employee => employee.Salary).Sum();
        }
    }
}
```

我们可以创建一个Employees类，并把计算总工资的逻辑放入其中：

```
public class Employees
{
    private List<Employee> employees;
    public int TotalSalary
    {
        get
        {
            return employees.Select(employee => employee.Salary).Sum();
        }
    }
}
```

虽然很多人反对创建这样的包装类，但在对集合的操作逐渐增多的情况下，就能显出这种做法的高明之处了。

4.1.4　惰性求值

使用迭代器时，我们经常会遇到一个问题：对一段代码反复求值。

比如，我们可能会用如下代码从一个文件中读取人名：

```
public class FileReader
{
    public IEnumerable<string> ReadNamesFromFile(string fileName)
    {
        using(var fileStream = new FileStream(fileName, FileMode.Open))
        {
            using(var reader = new StreamReader(fileStream))
            {
                var nextLine = reader.ReadLine();
                while(nextLine != null)
                {
                    yield return nextLine;
                    nextLine = reader.ReadLine();
                }
            }
        }
    }
}
```

稍后，PersonRepository会调用到上述代码：

```
public class PersonRepository
{
    private FileReader fileReader;
    IEnumerable<Person> GetPeople()
    {
        return fileReader.ReadNamesFromFile("names.txt")
                         .Select(name => new Person(name));
    }
}
```

而PersonRepository又会在代码中的其他地方被这样调用：

```
var people = personRepository.GetPeople();
foreach(var person in people)
{
    Console.WriteLine(person.Name);
}

Console.WriteLine("Total number of people: " + person.Count());
```

这样会导致两次读文件：打印名字时读一次，打印人数时又读一次。这是因为ReadNamesFromFile()方法是惰性求值的。

我们可以通过强制及早求值（eager evaluation）来避免这个问题：

```
public class PersonRepository
{
    private FileReader fileReader;
```

```
    IEnumerable<Person> GetPeople()
    {
        return fileReader.ReadNamesFromFile("names.txt")
                         .Select(name => new Person(name))
                         .ToList();
    }
}
```

4.2 “一等公民”和高阶函数

高阶函数可以接收其他函数做参数，或是返回一个函数。在前文中，我们已经见过很多以其他函数为参数的函数了。

函数（或者lambda表达式）在C#中是“一等公民”；我们可以把它用在代码中的任何地方，从而使用其他的语言实体。实际上，lambda表达式会由编译器转换成委托，所以它只是一种设计时对程序员可见的语法糖。

lambda表达式的语法让我们可以更容易地把函数当参数传递。

把函数作为参数传递时，唯一需要注意的是保证日后依然可以理解我们今天写下的代码。当我们大量运用Func()和Action()进行委托时，一不小心就会写出无法理解的代码。要免受此痛苦，一种方法是使用具名的委托描述函数的用途。

举例来讲，如果我们把下面这样的Func作为函数传递：

```
public class PremiumCalculator
{
    public Money CalculatePremium(Func<Customer, DateTime, Money> calculation)
    {
        // 计算溢价
    }
}
```

就可以定义下面的委托：

```
public delegate Money PremiumCalculation(Customer record, DateTime effectiveDate);
```

然后，用它代替上面代码中用到的Func：

```
public class PremiumCalculator
{
    public Money CalculatePremium(PremiumCalculation calculation)
    {
        // 计算溢价
    }
}
```

由于修改前后的代码并不等价，所以不能一一对应修改调用点，必须一次性地把所有调用点替换成上面定义的具名委托。

简化经典设计模式

如果把函数当做参数传递，可以极大地减少实现《设计模式：可复用面向对象软件的基础》[GHJV95]一书中某些设计模式的代码量。

在最近的一个项目中，我们需要与二三十个Web Service进行交互，我们想找到一种通用的方式缓存请求的结果，于是就用了如下的代码实现，这是一种装饰器（decorator）模式[①]的变体：

```
public class ServiceCache<TService>
{
    protected readonly TService service;
    private readonly ServiceCache cache;
    public ServiceCache(TService service, ServiceCache cache)
    {
        this.service = service;
        this.cache = cache;
    }
    protected TResp FromCacheOrService<TReq, TResp>(Func<TResp> service, TReq req)
    {
        var cached = cache.RetrieveIfExists(typeof(TService), typeof(TResp), req);
        if (cached == null)
        {
            cached = service();
            cache.Add(typeof(TService), req, cached);
        }
        return (TResp) cached;
    }
}
```

因为可以给`FromCacheOrService()`方法传递一个函数作为参数，所以无需再给`ServiceCache`添加一个抽象方法，也无需让每个需要缓存的`service`都去实现该抽象方法。我们可以像下述代码一样使用`SerciceCache`：

```
public class CachedPaymentService : ServiceCache<IPaymentService>, IPaymentService
{
    public CachedPaymentService(IPaymentService service, ServiceCache cache)
            : base(service, cache) {}

        public PaymentResponse GetPayment(PaymentRequest params)
        {
            return FromCacheOrService(() => service.GetPayment(params), params);
        }
}
```

① http://en.wikipedia.org/wiki/Decorator_pattern

4.3 状态最小化

函数式编程的另一个关键理念是，避免在应用程序中使用可变状态。

通过创建值（value），而非变量（variable）就避免使用可变状态。在函数式编程语言中，值在创建之后通常就无法修改了；换句话说，值是不可变的。

在面向对象语言中，要完全避免可变状态是很困难的，而且也不符合习惯。但我们还是可以降低修改状态的频率，提高代码的可读性。

比如说，在Ruby中散列表经常会这样用：

```
delivery_costs = {}
[:standard, :next_day, :same_day].each do |type|
  cost = DeliveryService.calculate_delivery_cost(delivery_address, type)
  delivery_costs[type] = "%.2f" % cost
end
```

在这段代码中，我们创建了一个叫做delivery_costs的变量，并在each循环中修改它的状态。

在这段代码中，修改delivery_costs的问题不大。但是如果定义delivery_costs的代码和填充它的代码不在一处，那就很可能会被delivery_costs状态的变化搞得晕头转向。

下面的代码通过使用reduce()方法封装了变化：

```
delivery_costs = [:standard, :next_day, :same_day].reduce({}) do |result, type|
  cost = DeliveryService.calculate_delivery_cost(delivery_address, type)
  result[type] = "%.2f" % cost
  result
end
```

如果愿意，我们仍然可以在代码的其他地方修改delivery_costs，但起码在创建和填充散列表的代码中无需再修改delivery_costs。

另一个减少可变状态的方法，是只在需要计算结果的时候才进行计算，而不是先进行计算，把结果[1]保存在字段中。

我经常见到风格如下的代码：

```
public class PaymentService
{
    private double monthlyPayment;
    private double yearlyPayment;

    public PaymentService(ExternalService externalService)
```

[1] http://www.markhneedham.com/blog/2009/09/02/coding-reduce-fields-delay-calculations/

```
    {
        this.monthlyPayment = externalService.CalculateMonthlyPayment();
        this.yearlyPayment = externalService.CalculateYearlyPayment();
    }

    public double MonthlyPayment()
    {
        return monthlyPayment;
    }

    public double YearlyPayment()
    {
        return yearlyPayment;
    }
}
```

在PaymentService的调用者调用MonthlyPayment和YearlyPayment之前，我们并不需要知道月付款和年付款是多少。

我们也没有必要在PaymentService中创建状态。

相反，我们可以保存externalService，并在外界请求付款值时再进行计算：

```
public class PaymentService
{
    private ExternalService externalService;

    public PaymentService(ExternalService externalService)
    {
        this.externalService  = externalService;
    }

    public double MonthlyPayment()
    {
        return externalService.CalculateMonthlyPayment();
    }

    public double YearlyPayment()
    {
        return externalService.CalculateYearlyPayment();
    }
}
```

这样修改的代码可能会多次调用ExternalService，但可以等到问题发生时再处理。

4.4　其他理念

接下来，我们看一看函数式编程中的一些其他理念。

延续传递风格

现在，我们可以使用延续传递风格（Continuation Passing Style，CPS），通过这种方法可以把部分计算作为函数传递给另一个函数。

下面的代码是延续传递风格的一个例子：

```
static void Identity<T>(T value, Action<T> k)
{
    k(value);
}
```

要调用上面的方法，可以用如下代码：

```
Identity("foo", s => Console.WriteLine(s));
```

上面的代码把计算的一部分——也就是打印语句——传给了`Identity()`方法，这样就将程序的控制权交给了这个函数。

下面是一个更有趣的例子[①]，我把这段`controller`代码转成了随后的CPS：

```
public ShoppingController : Controller
{
  public ActionResult Submit(string id, FormCollection form)
  {
    var shoppingBasket = CreateShoppingBasketFrom(id, form);

    if (!validator.IsValid(shoppingBasket, ModelState))
    {
      return RedirectToAction("index",
            "ShoppingBasket", new { shoppingBasket.Id });
    }
      try
      {
        shoppingBasket.User = userService.CreateAccountOrLogIn(shoppingBasket);
      }
      catch (NoAccountException)
      {
        ModelState.AddModelError("Password", "User name/email invalid");
        return RedirectToAction("index", "Shopping", new { Id = new Guid(id) });
      }

     UpdateShoppingBasket(shoppingBasket);
     return RedirectToAction("index", "Purchase", new { Id = shoppingBasket.Id });
  }
}
```

① http://www.markhneedham.com/blog/2010/03/19/functional-c-continuation-passing-style/

我曾在jQuery代码中见过这种用法，把处理校验成功和校验失败的函数作为回调，传给了其它函数。

```
public class ShoppingController : Controller
{
  public ActionResult Submit(string id, FormCollection form)
  {
    var basket = CreateShoppingBasketFrom(id, form);
    return IsValid(basket, ModelState,
      failureFn: () => RedirectToAction("index", "Shopping", new {basket.Id}),
      successFn: () =>
        Login(basket,
          failureFn: () => {
            ModelState.AddModelError("Password", "User name/email invalid");
            return RedirectToAction("index", "Shopping", new {Id = new Guid(id)});
          },
          successFn: user => {
            basket.User = user;
            UpdateShoppingBasket(basket);
            return RedirectToAction("index", "Purchase", new {Id = basket.Id});
                }));
  }

  private RedirectToRouteResult IsValid(ShoppingBasket basket,
                                        ModelStateDictionary modelState,
                                        Func<RedirectToRouteResult> failureFn,
                                        Func<RedirectToRouteResult> successFn)
  {
    return validator.IsValid(basket, modelState) ? successFn() : failureFn();
  }

  private RedirectToRouteResult Login(ShoppingBasket basket,
                                      Func<RedirectToRouteResult> failureFn,
                                      Func<User,RedirectToRouteResult> successFn)
  {
    User user = null;
    try
    {
      user = userService.CreateAccountOrLogIn(basket);
    }

    catch (NoAccountException)
    {
    return failureFn();
  }

  return successFn(user);
  }
}
```

这段`controller`的代码根据校验的成功或失败进行后续操作，所以，我同时传了成功和失败的延续。

在改写后的代码中，处理提交的主代码中不再包含`try/catch`代码块，这一点我十分满意，而且原来处理跳转和登录的代码混杂在一起，现在也分开了。

不过，总体上讲，这段代码修改之后和之前读起来并没有太大的不同。

代码修改之前，其逻辑是由上至下的，而修改之后，逻辑则是由左至右的。但这样读起来不太自然，所以修改之后的代码更为复杂了。

4.5 总结

日复一日地使用面向对象程序设计语言的害处之一，就是它会日渐渗透进我们的思考方式。而拥抱不同的编程范式，可以让我们以不同的方式理解问题。前文中的很多代码都只是集合转换而已，以转换思维看待集合，可以引导我们将其视为“一等公民”。以集合转换的方式思考，我们可能会发现当下的问题其实可以延展到更广的领域，甚至会发现有人已经为我们解决了问题。

本文中提到的技巧也可以用来简化设计模式。使用高阶函数让我们以一种不同的眼光去看待问题。学习新的编程范式，可以帮我们掌握更多解决问题的技巧。函数式编程则提供了很多解决老问题的创新方式。

Part 2 第二部分

测　　试

由 5 位 ThoughtWorks 员工撰文，探索了敏捷与技术的相关问题，其中涵盖了极限性能测试、JavaScript 测试和验收测试。

第5章

极限性能测试

5

Alistair Jones和Patrick Kua撰文

极限编程将一些常识性的原则和实践推向了极致。

——Kent Beck

在与客户合作的项目中，敏捷方法起到了关键作用。客户的软件系统通常都有很高的性能要求，但我们发现很少有敏捷文献涉及这一主题，而且现有的资料也无法回答“敏捷方法如何应用在性能测试中”这一问题。

我们归纳了20余个敏捷项目的经验，帮助我们将敏捷价值和原则[①]应用在性能测试中。我们已经形成了一系列成功实践，而且这些实践均已通过客户证明了其可行性，希望你也可以从这些实践中获益。我们把这套实践其称为*极限性能测试*。“极限性能测试”一词的灵感来源于*极限编程*（XP）[Bec00]。极限编程特别强调了能让敏捷方法行之有效的各项实践，这对我们的工作有很大的影响。

5.1 问题描述

一直以来，软件开发团队都很关心软件的性能问题。但过去几年里，问题焦点已经发生变化，不仅仅局限于在有限的硬件之上实现功能，而且还要更加关注负载增加时功能的扩展性。然而，关注点并未发生实际变化——在实现一系列功能的同时，我们还需要考虑软件是否如预期的那样工作。最近出现的云计算，也开始致力于使计算资源的成本清晰明了，另外我们也开始注意到大家对软件性能的关注。

5.1.1 分离性能测试的传统方式

如果软件项目中包括性能测试，我们通常将其视为作为一个独立的项目阶段，把它安排到开

① 如http://www.agilemanifesto.org所述。

发之后、产品上线之前。在《应用程序性能测试的艺术》[Mol09]一书中，Ian Moluneaux也描述了一个严格遵循该模型的流程。他将性能测试本身看做一个独立项目，由以下几部分组成：

- 捕获需求；
- 搭建测试环境；
- 事务脚本；
- 构建性能测试；
- 执行性能测试；
- 分析结果，报告以及重新测试。

这种类型的性能测试工程适合外包给外部供应商。除此之外，由于很多公司都有专门的性能测试团队，所以也可以根据需求为各个项目提供性能测试服务。以上方式的共同点在于，性能测试是一项独立于软件编写工作的活动，由不同的团队负责。

独立进行性能测试的原因如下所述。

- 性能测试被看做是一次验证过程，因此从逻辑上可与其他部署前的准备活动（比如用户验收测试）归为一类。
- 性能测试需要专门的技能，而且通常大家都认为，相比在一个综合团队中培养性能测试技能，将性能测试技能集中在专门的团队中，效率会更高。

5.1.2　极限编程和敏捷软件开发

在过去的十年里，敏捷方法越来越受到软件开发团队的欢迎。该方法的特点包括：迭代和增量开发，客户的密切参与，以及定期排列优先级和制定计划。因为敏捷团队致力于使软件可持续发布[①]，所以测试工作量包括在了交付功能的成本中。敏捷团队整合了测试的能力，以便可以在开发的同时验证功能，并以交付的被测功能来衡量生产力。这与传统模式中由独立团队实施测试的方式形成了鲜明对照。

除了这些管理实践，极限编程还推荐了许多辅助工程实践。这些实践提高了开发过程中的规范性和质量，确保了交付运转正常的软件，使迭代式开发切实可行。

然而，极限编程并没有专门的性能测试，而敏捷社区对于如何将性能测试整合到开发过程中也没有明确的指导。我们在早期的敏捷项目中发现，性能测试是作为独立的活动运行于核心开发团队之外的。如今，性能测试依然保留在迭代开发过程之外，在开发之后、发布之前按照瀑布模型执行。

---

① 详见《解析极限编程》[BA04]。

5.1.3　分离性能测试的不足

分离性能测试的普遍弱点就是测试时间太晚。尽管很多项目都希望尽早进行性能测试，但实际上很难在大部分开发工作完成之前安排性能测试。如果在项目中性能测试开始的时间太早，软件还并不完整，测试结果也没有意义。而另一方面，如果测试时间太晚，测试出现的任何问题都会导致大量返工，很可能导致发布延期。有一种也许可行的方法是将性能测试分解为小块，将这些小块分布在整个项目的生命周期中。但是，如果安排专门团队负责这些小的测试块，每次要开始性能测试时会造成很大的开销，而且在多个团队之间安排任务也十分麻烦。

此外，另一个需要考虑的方面是性能测试需要深入理解被测系统。

- 性能测试的设计必须准确反映系统如何使用。
- 必须针对系统中正确的技术组件执行测试。
- 当测试没有如预期工作时，需要查找原因。
- 诊断性能瓶颈需要了解系统架构和软件设计。

如果性能测试团队独立于开发团队之外，就需要花费时间学习必要的测试情报，或者向拥有相关情报的开发团队寻求帮助。这部分工作量是除性能测试本身外额外增加的，对于复杂系统来说会花费大量时间。当开发团队本身处于交付压力下时，性能测试团队很难获取到必要的测试情报，这会给性能测试工作的成功带来风险。

从项目管理角度来说，很难决定分配多少时间和资源给性能测试。性能测试团队需要预先成立，或者在确认是否有潜在的性能问题之前就预留出测试时间。如果一开始的性能表现良好，则可以将计划用于性能测试的资源转到开发更多的功能上，但在这时改变方向已经太晚了，因为性能测试团队并没有软件开发的经验。相反，如果一开始就遇见很多性能问题，用于性能测试的时间会超过预期，并导致很多项目延期。此时最好的管理办法，只能是作出基于经验的预测。

总而言之，很难在项目中找出正确的时间，安排单独的性能测试，这会带来很大的沟通开销，同时也很可能造成资源浪费或者项目延期。

5.2　另辟蹊径

在近期带有性能要求的软件开发项目中，我们尝试了一种不同于分离性能测试传统模型的方式。借助已有的敏捷和极限编程经验，我们想出了将敏捷方法的原则应用在性能测试相关领域的方式。我们提出了性能测试与主要开发工作融为一体的原则，并得出了一系列可行的实践。这些实践非常成功，我们希望把这一方法推广到其他项目中。

5.2.1 独立的多功能团队

我们建议独立团队应该同时负责软件开发和性能测试。这种团队组成与典型的极限编程团队类似，团队成员应专长于业务分析、开发以及测试。

与不负责软件性能的开发团队相比，从事性能测试必须还要掌握一些额外技能，但我们不建议团队中指派专员负责性能测试。相反，很多团队成员除了自己已具备的技能之外，还应当具备性能测试技能。我们发现开发和测试人员通常都已掌握测试技能，或者也能够轻松地学会测试技能。

建立一支这样的团队有何意义？首先，这样能消除性能测试和开发团队之间的物理距离；测试和开发应在同一支房间里进行。然而，都在同一房间并不意味着就是一支团队。当需要性能测试时，不应当总是由一支人来做。或者说，不应该只由团队中的某一部分人做性能测试；团队中所有开发和测试人员都应该定期、积极地担负起测试任务。

一支典型的敏捷开发团队可能如图5-1所示的一样，他们旁边是一支独立的性能测试团队。我们建议推倒两个团队之间的墙，将他们整合起来，如图5-2所示。



图5-1　单独的性能测试团队

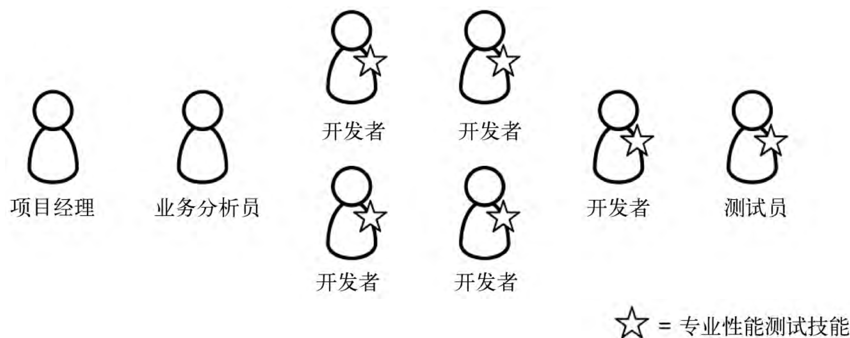


图5-2　整合后的开发与性能测试团队

我们建议具备性能测试技能的团队成员和其他人结对编程，这样他们的专业技能就能够传授给其他的团队成员。当很多人都具备了专业的性能测试技能时，因为某一支人无法工作而导致性能测试无法继续下去的风险就会大大降低。在项目后续性能测试中，团队工作也会更为灵活，可以根据需要灵活调整工作重心。

整合后的团队沟通成本降低，知识传播迅速，并且能培养一种乐于分享的文化。

5.2.2　描述需求

我们发现撰写用户故事（user story）是一种捕获性能需求的有效方式。敏捷团队习惯用标准结构书写用户故事，那就是"为了……，作为……，我想……"的模式。这种统一的模式对于明确表述出每个故事的动机和使用者有着明显的优势。对于性能测试，我们发现增加一些"如果……"的语句，能够强调特定的目标要在怎样的条件下满足。

有两种故事是和性能问题密切相关的。第一类故事描述了系统自身的需求，比如在特定的场景下支持特定的负载。当提到性能需求时，大家通常都会理解为这类故事。下面是一个例子：

> 为了投资者能够在业务增长时获得高质量的体验，
> 作为运维经理，
> 如果有10 000个登录用户查看同一个投资组合价值页面，页面每2秒就会刷新一次，
> 我需要在0.2秒内能渲染出这个页面。

当团队开始实现这个故事时，他们需要根据故事的描述度量系统的性能，而且当有任何性能改进的要求时，还要重新进行度量。度量要求通过高质量的测试来完成，而创建这些测试可能要耗费很大的精力。我们发现将创建测试的工作和提高系统性能的工作分开来做很有效。

这就引出了第二类性能相关的故事：实现测试的故事。实现测试的故事描述了系统的项目投资者想要如何度量系统。这里有个例子：

> 为了能够决定是否需要进一步的开发工作，
> 作为运维经理，
> 如果模拟至少10 000个登录用户负载，
> 我需要一个可重用的性能测试，用以度量投资者操作的响应时间。

在测试实现故事和系统性能故事之间有明显的依赖：测试实现故事应当先完成。对于一个复杂系统，可能有必要先完成一些测试实现故事，构建一系列用于度量和提高性能的工具。

将测试实现和系统性能分开是很合理的，有如下几条原因。

- 测试实现故事可以单独完成。即使还不清楚系统的性能目标，每个故事各自都对度量性能很有价值。

- 能够更容易地估计进度，识别出是什么工作占用了开发时间。这种清晰的划分也可以很方便地跟踪用于构建测试和提高性能分别花费的时间。

性能故事与传统的功能故事不仅结构相同，二者还有许多共同特征。因此，性能故事可以与项目中其他的用户故事一样，遵循同样的工作流程：首先创建并分析，然后实现并验证。性能故事被放在同一个产品功能列表中，等待分析和划分优先级。

足够小的工作单元

一个好的用户故事的特点就是小[1]。根据我们的经验，在同一个项目中，性能测试工作通常要比典型的功能故事花费更长的时间，所以我们会尽可能得将它们拆分成足够小的故事。小故事会带来更快的反馈，这样，当我们有足够多的信息来判断多久能完成这些故事时，就更容易对计划进行调整。

很多用于划分性能故事的技巧，都与划分功能性故事[2]的方法没有太大区别。

可能有很多不同的场景需要测试性能，特别是在不同负载分布下的场景。相比于在一个性能故事中设置所有场景，为什么不先从实现最简单的场景开始（比如，一个稳定的后台负载），然后在后续的故事中增添更多复杂的场景（比如，一系列因市场活动带来的突发峰值），循序渐进地增强原有的测试呢？

技术调研，是一项在限定时间能完成的调查研究，用来回答特定的技术难题。技术调研也可以用于性能测试，因为性能测试同样有很强的不确定性。举例来说，“我们现在的性能测试工具是否能够产生出足够的负载来模拟这个场景？”这个问题就很适合通过技术调研来解决。完成了技术调研后，后续的故事则风险较低，也更容易进行评估和计划。

如果需要用复杂的可视化方式诠释测试结果，也可以先实现一个简单的可视化描述，之后再增强其功能。比如，先绘制出一个变量，然后再用更具有鉴别性的数据丰富可视化描述。

5.2.3 设定计划与排定优先级

因为将由同一组人实现性能故事或开发系统功能，因此所有故事都应该放在同一个待办故事列表（backlog）或者说产品功能列表中。性能测试可以与功能开发一起排定优先级，这对于项目的投资人来说是非常有价值的。也许在第一次发布中，性能还不是很重要，因为用户数量预计会比较少。在这种情况下，投资者会选择将大部分性能相关的工作推迟到后面的发布中来做，而用户故事则能更为清晰地进行这种取舍。5.3.1节内容可以帮助极限编程的客户排定性能故事的优先级。

---

① 即《用户故事与敏捷方法》一书中“INVEST”原则中的S原则。

② 详见Rachel Davies在http://agilecoach.typepad.com/agile-coaching/2010/09/ideas-for-slicing-user-stories.html上的描述。

有了接下来要讲的自动化测试流程和持续性能测试，前面实现的测试不需要额外代价，就能在整个项目的过程中不断执行，这样性能相关的工作就不会浪费——即使测试实现时，软件还远未实现。一个迭代可能同时包含了性能故事和功能故事。图5-3展示了性能故事如何与功能故事在整个项目的过程中同步进行。

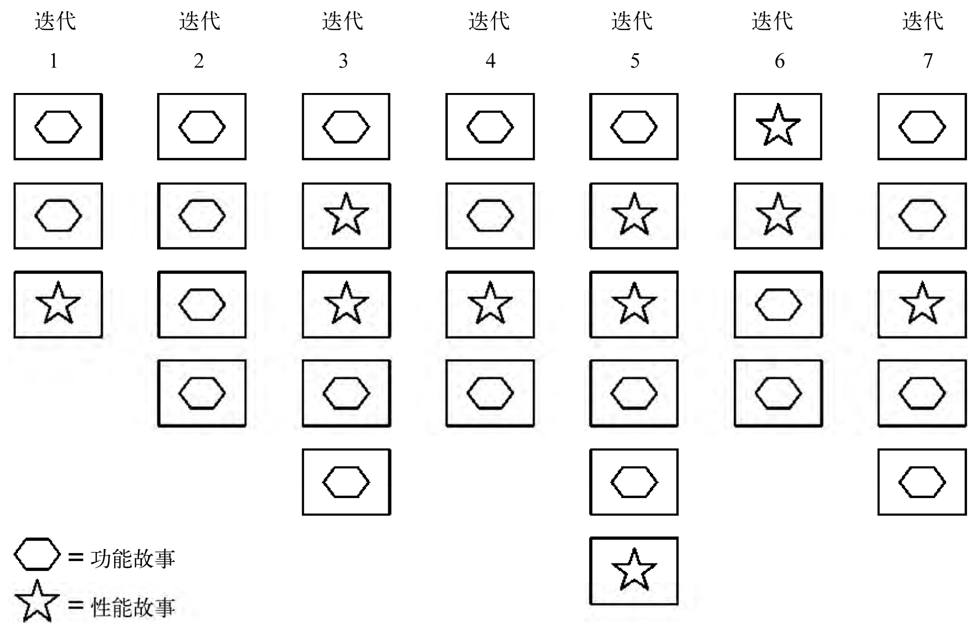


图5-3　迭代中的性能故事

排列功能故事的标准同样适用于性能故事：能交付最大价值的故事应当优先着手，同时高风险的故事也应当最先开始，以减少不确定性。应用这一标准到性能测试时，意味着我们应当优先计划能交付最大业务价值场景的测试，同样的，能够验证高风险技术决定的测试也应当先开始。比较性能故事和功能故事优先级时，我们应当考虑到，在基本的架构通过性能测试验证之前，会有大规模重做的风险。因此，应当在大量功能故事开始之前，先做一些基本层面的性能故事。

5.2.4　实现性能故事

一个典型的用户故事墙如图5-4所示。性能故事的生命周期应该与功能故事类似，它们也应当历经故事墙上所有的状态。

在开发功能时，每个故事都有验收条件，在开发之前，就已在适当的时间定义。这些验收条件定义了一个故事如何算是“完成”；在开发完成后的一个阶段中（QA阶段），验收条件会由开

发人员之外的人来验证。这道程序同样适用于性能故事；在性能工作开始之前，大家就应为验收条件达成一致，再在完成后独立验证（由某位没有参与性能故事实现的人来进行）。

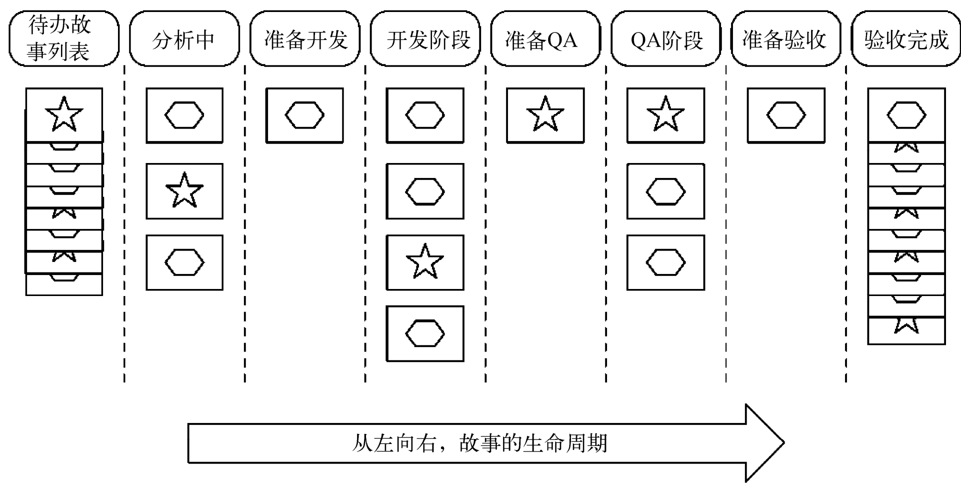


图5-4 典型的用户故事生命周期

对于测试实现故事，验收条件应当检查该测试是否的确能给系统施加预期的负载，而且测试的结果是可量化的。对于系统性能故事，验收条件应当是能在适当的控制条件下执行测试，并正确诠释测试结果。

5.2.5 展示与反馈

性能故事应当包括在团队的常规项目展示中，这样使性能测试工作更清晰可见，为项目投资人排定将来的性能测试工作提供更好的信息。性能测试的图形和图表非常适合用于项目展示。

5.3 极限性能测试实践

与独立的性能测试相比，极限性能测试面临着一些新的挑战，因此需要一些不同的实践支持这种新的方式。同时，极限性能测试也为提高性能测试本身的有效性和效率提供了实践契机。本节内容描述了实施极限性能测试的团队中一些行之有效的实践。

5.3.1 性能负责人

在极限编程中，极限编程客户（XP Customer）这种角色会基于业务价值对事务进行排序。担任该角色的人通常来自业务领域，因此他没有技术背景，往往无法考量性能成本。为了实现扩

展性，Google、Twitter、Facebook等网站都付出了不菲的代价。当无法持续得知由性能问题所带来的成本信息时，极限编程客户会继续将构建新功能的优先级排得更高，而不会为满足性能需求做任何投资。

性能负责人作为极限编程客户的有效补充，并非替代后者，它的职责描述如下。

- 让其他人了解性能问题：业务人员通常不懂各种不同的性能指标，包括延迟、响应时间、吞吐量，等等。性能负责人和其他角色一起工作，使其了解每个性能指标的含义，教会他们如何做取舍。帮助其他角色了解要在某一项指标上达到特定的等级，需要哪些额外的工作量。
- 负责排定优先级：忽略性能测试和调优，业务随之就会出现风险。性能负责人需要指出忽略软件系统主要性能指标所带来的风险。将新特性的增量和迭代交付与性能测试仔细衡量后排定的优先级，将是非常理想的成果。
- 切实制定性能所到达程度：一定要为各性能指标定义相应的目标值，以便了解性能测试所需花费的工作量。10个用户使用的应用程序，必然与互联网用户使用的应用程序有天壤之别，必须有人负责为每个性能指标建模。
- 定义应急策略：因为觉得风险较低，业务人员有时会决定放弃为性能调优投入资源。然而，如果他们作出这样的决定，必须有人决定应用如何优雅地降级。

5.3.2　自动化部署

所有软件项目都能从自动化部署中获益。自动化部署避免了人工失误，保证了一致性，提高了发布频率。如果遵循了极限性能测试，自动化部署就变得更有价值。在采用了独立性能测试的软件项目中，在性能测试环境中部署应用程序花费了很多时间。通常部署过程本身就很耗时，另外，性能测试环境平时极少用到，所以还得花大把时间保证正确的环境配置，并且还要检测应用的功能是否符合预期需求。

在极限性能测试中，自动化部署通常发生在每次性能测试运行之前。这样会减少升级应用的开销，而且能及时反馈每次代码改变后所带来的影响，另外也便于发布新版本，运行测试及比较结果。

用来做部署的技术通常都是一些脚本，能够将软件包传输到合适的服务器上并启动安装。适用于生产环境的硬件在性能测试时必不可少，这通常意味着比开发环境中用到更多的应用服务器；自动化脚本应当包括同时升级所有的应用服务器。现实中硬件设备一般都最有可能在公司网络的外部，也可能会在产品数据中心。如果是这种情况，那就需要更完备的脚本，使之能够跨过公司网络和产品数据中心之间严格的安全屏障进行操作。如果可以利用按需计算资源（使用公有云或私有云时），自动化部署还应当包括自动获取合适的计算资源以运行应用。

除了安装应用，部署脚本还应当安装产生性能测试负载的代理，用以运行性能测试。手动安装这些负载产生代理，而后却置之不理是不可行的：随着时间推移，代理的配置会变得不一致，届时将成为维护的噩梦。可以通过自动化安装尽早杜绝这些问题，准备一组干净一致的代理开始测试。

5.3.3 自动化分析

性能测试工具都以日志文件或报告的形式产生结果。必须对每次测试运行结果进行分析，针对“这样的性能是否可以接受？”这种问题进行回馈。我们推荐将测试结果的分析自动化，这样就不再需要人工来回答关于测试的关键问题。如果现有的性能测试工具有分析能力，则可以将分析过程自动化。否则，就要根据应用特定的需求写一些程序用以分析结果。

相比独立的性能测试的测试量，极限性能测试的测试量要大得多。这样也产生了更多的日志文件，如果没有自动化，分析这些日志的成本将相当高。

自动化还使得分析结果更加一致。一旦写好脚本，它总会以同样方式来诠释测试结果。人工诠释更容易犯错误，可能漏掉问题，导致风险，或者会浪费时间，仅仅捕捉一些莫须有的异常情况。减少人工错误最好的方式就是自动化。

5.3.4 结果仓库

性能测试的结果是很有价值的。结果仓库（result repository）会存放并组织性能测试结果，这样就可以在应用程序的整个使用过程中获得最大的价值。

每次性能测试运行后，将捕获完整的原始结果和参考数据，如下所示。

- 测试执行的时刻与持续时间。
- 测试中执行的部分场景。
- 测试执行所针对软件的具体版本。（应当要能追溯到代码版本控制中的特定版本，只有日期或版本号是不够的。）
- 应用部署环境的详细情况，以及用来产生负载的代理的详细情况。硬件和环境配置会随着时间变化，这些变化与测试结果都很重要。

结果仓库应当同时记录下原始的结果数据和自动化分析的结果。保留原始数据很重要，因为如果随着项目的进行，分析能力不断增强，最后可以从重新分析历史测试数据。此外，保存下分析结果也很有用，因为分析本身也要花费时间，如果仅为找到历史测试结果而进行重新分析，则会非常麻烦。

构建良好的结果仓库可以支持基于历史数据的分析。它可以帮我们解答这样的问题：“我们

的应用是否总是表现出这样的行为？”

有很多不同的技术可以实现结果仓库。在过去的项目中，我们利用持续集成服务的手动产品库来存储测试结果，因为这样很容易追踪到测试执行的软件版本。结合该产品库，还可以在一个简单的关系数据库中记录下历史趋势，其中包括对底层原始数据的引用。我们不推荐在源码控制系统中保存测试结果，这些工具并不是为了存储这类数据而设计的，它们很快就会被海量的原始数据淹没。

在传统的性能测试方法中，经常通过邮件分发结果，查找历史数据意味着在收件箱中苦苦搜索。通常只有专门的性能测试团队才能访问原始数据。通过实现一种规范的结果仓库，我们可以开放对结果数据的访问，继而能从测试结果中攫取更大的价值。

5.3.5 结果可视化

频繁运行性能测试会产生大量的数据。在一次12小时的运行结果中，一行一行地查看数据非常困难，很容易出错，而且具有主观性。在处理这些数据时，要沟通描述清楚变化的趋势或性能特征就更加困难了，通常需要有专人负责解释。

可视化对于理解性能测试执行的结果很重要。视觉上不寻常的特征要比一大堆数字中的大号字体更吸引眼球。我们倾向于使用图形，因为大多数性能测试都有某种基于时间的自然特性。

要避免每次执行测试都产生一个单独的图形。制图数据太多会使图表更难理解。用一个图表关注一组更小的、特定的元素结合，创建多个图表来分别可视化不同的元素。第12章针对如何更有效地将数据可视化提出了更多的指导。

5.3.6 自动化测试流程

要想成功地启动、执行，以及停止测试运行，就必须遵循一系列步骤，也就是测试流程。下边就是一个运行性能测试的步骤范例。

(1) 部署应用。
(2) 等待部署完成（频繁的部署是一个异步操作，需要等它真正准备好再处理请求）。
(3) 启动负载生成器（如果需要高负载，可能需要在隔离的硬件上运行多个负载生成器，那这一步骤就必须要协调好所有的负载生成器实例）。
(4) 等待系统预热（通常当应用刚刚启动时，运行结果不具有代表性。普遍采用的实践是在主要的性能测试开始之前，在系统预热期间先向应用程序发送少量的请求）。
(5) 增加负载（负载程度由测试计划决定）。

(6) 等待经过一段测量区间（同样，不同的负载程度所持续的时间由测试计划决定）。

(7) 重复第(5)步、第(6)步，直到完成测试计划（测试计划可能包括一系列增加或减少负载的步骤）。

(8) 停止负载生成器。

(9)（可选步骤）停止应用程序，释放所有临时分配的硬件。

(10) 收集性能测试结果。

一个完备的性能测试工具会自动调整负载以匹配测试计划。然而，仅从负载生成器的角度来看，性能测试工具是无法对整个过程进行调整的。考虑测试流程时，我们发现要考虑整个测试的过程是很重要的。测试流程应当包括任何需要对应用程序做的处理，比如等待应用程序启动，等待预热完成等。要得到好的测试结果，并不存在只有性能测试人员才能知道的特殊技巧。任何特殊技巧都应当共享，成为测试流程自动化的一部分。

测试流程完成了自动化后，测试就能够在无人监控的情况下运行。一般来说，没有必要盯着看性能测试的运行。一旦触发测试，团队应当信心十足，相信测试能够在不中断的情况下顺利完成，而且测试结束之后也不会产生什么问题。

5.3.7　健全性测试

性能测试的本质特点意味着反馈周期很长。单个测试的运行时间短则1小时，长则一周。任何一个小小的错误都会导致整个测试无效，而测试所花费的时间还不如用在分析或者运行其他测试上。健全性测试扩展了敏捷中快速失败（failing fast）的理念。健全性测试以缩小后的频率或规模来运行性能测试的整个过程，目的就是能够测试性能测试过程本身。一个成功的健全性测试可以尽早发现性能测试过程中的错误。健全性测试通过保证测试执行的有用性和有效性，在代价不菲的测试环境中，用最小的成本将收益最大化。

如果没有验证，自动化过程很容易出错。性能测试中一个常见的问题是部署或应用程序配置错误。针对这种情况，可以进行简单的健全性测试，即在很短的时间内，在该环境中运行一次自动化测试，仅仅是为了验证系统是否成功部署，能否表现出测试的预期行为。这种测试对于没有任何自动化测试覆盖的系统部署很有用。如果应用程序无法工作，性能测试是没有意义的。

因为有许多独立的组件和不同的应用程序流程，性能测试过程变得非常脆弱。复制或打包测试执行结果往往就是个问题源，常有各种小问题出现，比如日志文件散落在不同机器的不同目录下，或者无法自动创建最终的结果报告。如果没有输出，性能测试就白白浪费了。在这种情况下，一次精炼的自动化健全性测试能够验证性能测试过程是否有适当的产出。比如在一个客户端中，测试执行之后的构建流程就包括创建结果的打包文件，该文件包括展示测试结果可视化图片，以及测试运行环境的详细描述。等待很长时间，结果却发现这个自动化过程失败，的确非常浪费时

间。健全性测试为我们节省了很多时间。我们会用脚本生成所有打包过程成功执行所必需的文件，它可以帮我们更快地找出定制脚本中的微小错误。

通过可用性测试检测内存泄露则困难得多，但对基于VM的程序，这也不是不可能，只要能确定一个小数据集的最大内存占用即可。随后，用受限的最大堆容量启动应用。

5.3.8　持续性能测试

持续性能测试将持续集成[DMG07]进行了扩展。传统的持续集成用以验证应用构建，并满足其功能性需求。对于测试来说，则为开发团队提供及时反馈。持续性能测试为持续集成增加了一个维度，保证应用满足性能需求。另外，它能尽快标示出新的性能问题。

持续性能测试构建在自动化部署、自动化分析和自动化测试流程这些实践之上。此外，持续性能测试在应用的构建流水线上又增加了一个额外的步骤，即能够部署应用的最新版本，执行性能测试，并验证应用程序以满足性能目标。如果性能目标没有满足，构建步骤就会失败，持续集成会以常规方式警告开发团队。此外，还可以根据以往的测试结果自动设置性能目标，如果和以往结果相比，性能有一定程度的降低，构建步骤就会失败。

持续性能测试主要的好处在于，一旦代码发生改变，很快就能发现代码变化导致的性能降低。这样，修改代码的开发人员可以在忘记代码变动之前，对问题进行调查并修复。如果是设计缺陷，团队可以避免进一步开发更多有同样问题的功能。

5.3.9　规范的性能提升

如果性能测试表明应用程序满足性能需求，就不需要进行额外工作来提高性能。相反，如果同样的测试表明系统无法满足性能要求，这些测试就需要扮演第二种角色，帮助诊断问题并修复。

要诊断和提升性能时，我们推荐一种高层次的规范，该规范要遵循一种流程（受到《科学方法》[Gau02]）的启发。没有这个规范流程，开发团队通常都会忘记在不同的测试执行时都做过哪些修改，继而也就无法确定所作修改产生的影响，并最终做出一些对软件不必要或无益的修改。

我们建议将规范的性能诊断和增强构造为一系列明确定义的循环周期。一个合适的性能提升流程如图5-5所示。

关于这个流程，重要的是在形成假设和对应用程序做长期修改之间要明确区分。没有区分，就无法确定代码修改是否真的提高了性能，而且也有可能带来风险，比如做出的修改引入了不必要的复杂性，甚至降低了性能。

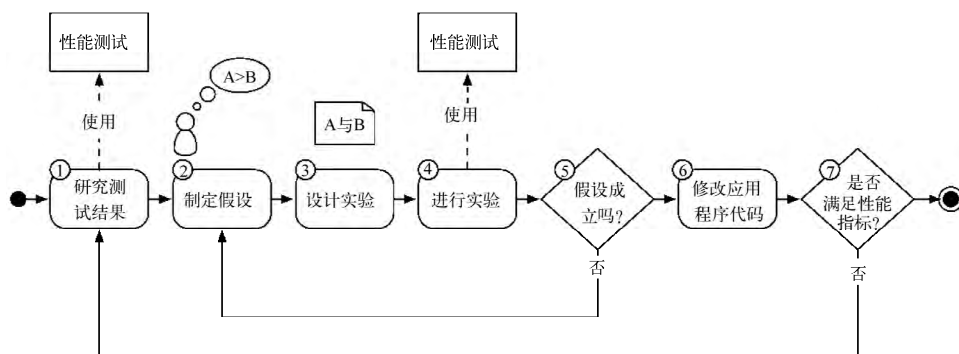


图5-5　性能提升周期

过去，我们看到团队同时做了很多性能相关的修改，然后仅用一个测试验证所有修改产生的影响。结果自然无法令人满意，因为这样做无法确定单个修改的影响。

要遵循规范的性能提升可能看起来需要很多工作，但却可以使极限性能测试的其他实践变得很容易。特别是，能将这个周期中的大部分步骤自动化，可以很容易地重复多次运行测试。这样一来，就可以在实际测试中隔离每种单个变化并检测所带来的影响。

5.4　这对我们有何帮助

那么，极限性能测试对我们有何帮助?

5.4.1　更好的性能

性能问题越早发现，就越容易修复。像持续性能测试这样的实践，能够迅速发现破坏性能目标的小改动。尽早发现问题能避免了一些开销，也不用去追踪一系列可能引起问题的改动。这样就可以把精力投入到寻找更好的解决方案上。

5.4.2　更低的复杂度

当性能测试团队和其他开发工作融合得更好，可以减少重复的工作量。把既有的自动化测试工具重用为性能测试的测试夹具，意味着需要编写和维护的代码更少。任何应用接口的变化只会影响其中一份代码。

正如测试驱动开发[Bec02]提高了代码设计一样，性能测试驱动开发会优化系统设计。极限性能测试中着重强调自动化，会让开发人员在应用、配置及部署过程中考虑用脚本实现，同时也使得其他任务（比如系统监测）变得容易许多。

5.4.3　更高的团队效率

我们发现开发和性能测试之间的连续性大有裨益。在传统的模式中，完成的应用会交给一个单独的性能测试团队，而通过整合的方式，可以在同一个团队中完成两项活动。我们有了关于应用程序的具体知识，这将有助于性能测试。在独立的性能测试方式中，则必须保证性能测试团队在长流程中的某个特定时期随时待命，但这样做风险很高，如果项目期限要调整，就会非常不灵活。

5.4.4　更合理的优先级排定

极限性能测试给了项目的相关人等更大的选择空间，他们可以决定性能测试投入的资源、何时投入资源以及获取决策所需的信息。这不需要测试团队的介入（如果需要他们介入，时间多久），相关人等可以决定更细粒度的性能测试，并和功能开发一起排定优先级。此外，还可以基于已有的性能测试结果决定要完成多少性能测试故事。如果在项目早期达到了一些性能目标，并且不需要进一步提高，那么额外的资源可以用于开发功能。

5.4.5　开启持续交付

在《持续交付：发布可靠软件的系统方法》[HF10]一书中，Jez Humble和David Farley描述了如何定期地、顺利地将可工作的软件发布到产品环境。极限性能测试可以看做是持续交付的支持实践。自动化的性能测试扮演了类似于自动化回归测试套件的角色：降低了验证软件是否可发布的成本，因此可以更频繁地发布软件。

5.5　总结

我们在实际项目中都验证过本章所述的这些实践，并得到了很好的结果。极限性能测试完全可以为更多人所用。

极限性能测试也并不是在每个团队、每种场景都适用。如果想要采用极限性能测试，开发团队应当已经实践了敏捷方法，应当具备极限编程倡导的很强的工程技能。而且，最好从一个有明显性能要求的项目开始，这样才可以从这些实践中获得最大收益，让团队的所有成员都能获得大量的经验。

我们期待有更多的团队采用极限性能测试。期望大家能够改进这些实践并应用到自己的项目当中。更重要的是，我们想看到所有团队都能够创建出许多高性能的应用程序！

第 6 章

测试驱动JavaScript



Brian Blignaut和Luca Grulla撰文

过去的几年里，Web已经从简单展示静态内容的平台变成了一个可以交付富互联网应用的平台。大量的DOM操作和Ajax回调是每个Web2.0应用的基础。JavaScript正是使得这一切成为现实的非官方的标准语言。

但不幸的是，在测试方面，JavaScript代码没有受到与后端代码同样程度的关注，这就导致了客户端的代码难以维护与增强。对用户而言，这也带来了不一致并且易出错的体验。

本文会讨论在大型Web项目的背景下，如何编写和测试客户端的JavaScript。

6.1 JavaScript 的复兴

出现伊始，JavaScript就被大多数开发者当做二等语言对待。尽管过去几年里，JavaScript的代码无论是规模还是复杂度都有了极大增长，但其编码实践发展速度远不如其他语言。

此外，随着Web2.0的爆发，涌现出不少JavaScript库，它们定义了客户端JavaScript开发的新方法。像jQuery[①]这样的库帮助开发者解决一些Web开发中的困难问题（比如跨浏览器的问题），jQuery的某些特性可以提高开发者的生产力，比如DSL风格的选择器和高级动画。但这种额外的威力却也增加代码的复杂性：连贯接口API和高级选择器的混合，让代码风格更加简洁，却也给阅读和演进带来了困难。

现代JavaScript引擎的附加能力和HTML 5的高级特性给了用户体验设计师和开发者更好的驱动力，可以打造更丰富的用户界面和交互功能，释放Web的潜能。一些高级特性(比如本地存储支持的离线使用应用程序）把Web应用程序推向富客户端应用程序的康庄大道，使浏览器成为一个托管环境。

① http://jquery.com

因此，我们需要一些可靠的工程实践和方法，保证这些复杂性的可控性，支撑代码库的有机增长，规避混乱的JavaScript代码风格。同样，我们也需要一些实践，帮助我们规避退化，以便在发布之后继续演进代码。

单元测试实践已众所周知，它能帮助我们达成内部软件质量，降低缺陷率。单元测试的粒度很细，它关注某个同协作者隔离开来的特定组件。其中的关键点在于隔离：如果可以关注某个特定行为，而忽略系统额外的复杂性，测试就会非常具体；而一旦有错误，也非常容易识别遭到破坏的部分，对它进行修复。

6.2　当前 JavaScript 的处理方式与问题

客户端JavaScript的特点是事件驱动：用户与屏幕上的某个组件交互（例如单击按钮），应用执行动作，然后向用户显示新的或其他信息。

在数量庞大的JavaScript代码库中，扮演系统入口点角色的事件处理器常常承担了过多的责任。同一个回调函数里，数据处理、DOM转换以及服务器端的Ajax调用通信都混在了一起。

代码也趋于在底层操作，但缺少围绕领域的建模和更基础的功能层。JavaScript最终与DOM紧密耦合，以致于测试JavaScript层唯一的方法就是使用HTML夹具。由于DOM操作通常用以展示一些Ajax调用结果，所以JavaScript代码最终不仅与DOM耦合在一起，也与服务器耦合在一起，服务器必须由整个应用程序栈提供数据。

这种情况会导致两个问题。

首先，代码只能进行黑盒测试，对Web应用程序而言，这意味着要在浏览器级别做验收测试。测试特定的应用场景时，基于浏览器的验收测试极具价值，但如果只测试应用程序里某些单独的功能区，这种做法就显得缓慢而脆弱了，因此这不是测试JavaScript的理想方法。相对于我们要做的事情而言，这种做法层次太高：如果验收测试失败了，它可能是由应用程序中的任意一层的问题所引起，因为验收测试一般是将整个应用程序当做一个整体测试。

其次，这会导致代码具体化。如果代码没有清晰的设计，很难演进或增加新功能，继而会引入重复和不必要的复杂性。

6.3　分离关注点

我们来看看代码。（从现在开始，我们会使用jQuery以及它的标准符号$作为JavaScript的核心库。）

JavaScriptTesting/loginPage.js

```
function LoginPage() {
    this.setup = function() {
        $("#loginButton").click(this.login)
    },

    this.login = function (e) {
        var username = $("#username").val();
        var password = $("#password").val();

        if (username && username !== "" && password && password !== "") {
            $.ajax({
                url: "/login",
                type:"POST",
                data:{username:username, password:password},
                success: loginPage.showLoginSuccessful,
                error: loginPage.showLoginError
            });
        } else {
            loginPage.showInvalidCredentialsError();
        }
        e.preventDefault();
    },

    this.showLoginSuccessful = function() {
        $("#message").text("Welcome back!");
        $("#message").removeClass("error");
        $("#message").fadeIn();
    },
   this.showInvalidCredentialsError = function() {
       $("#message").text("Please enter your login details");
       $("#message").addClass("error");
       $("#message").fadeIn();
   },

   this.showLoginError = function() {
       $("#message").text("We were unable " +
                          "to log you in with the details supplied");
       $("#message").addClass("error");
       $("#message").fadeIn();
   }

};

$(document).ready(function() {
    var loginPage = new LoginPage();
    loginPage.setup();
});
```

这是一个非常简单的登录脚本。它会验证输入，然后尝试以用户提供的认证信息进行登录。上述业务逻辑的代码完成了下述三步：

- 用户输入验证；
- 根据验证结果确定后续步骤；
- 尝试登录。

代码的其他部分处理的是展现逻辑和集成逻辑。

就目前的情况而言，如果忽略单一职责原则[①]（Single Responsibility Principle，SPR），上述代码的可读性就会相当好。但如果验证逻辑突然变得更复杂了，我们该怎么办呢？或者说，业务负责人决定展现给用户的消息需要更多的动画和特效，又该如何是好呢？就当前的代码而言，实现这些功能只会导致混乱的代码。如果不设置完整的HTML夹具，端到端地运行，就几乎无法测试。虽然这无疑也是一种方法，但我们无法保证使用的HTML夹具可以代表产品环境中真实的状态。这就意味着可能最终导致误报应用程序的状态。

那么应该如何解决这个问题呢？

与其他语言一样，要编写可读、可测和可维护的JavaScript代码，分离关注点扮演着重要的角色。

如果能够识别和分离代码中不同的角色和职责，就可以创建组件，通过交互实现我们想要实现的特定功能，取代那些混乱的底层指令集。组件就会提升封装性并强制SRP，它通过创建好的API，使代码变得灵活，易于修改，使对象间的契约得以规范。

因此，我们要尽量将组成不同功能的代码区隔离开来。通过隔离这些不同的域，就能够使用标准的单元测试方法测试应用（比如mock和stub[②]），而无需复杂的HTML夹具。这种方式还可以帮我们确保代码仍然是松耦合的，而且也能从重构中受益。

仔细看一下上面的例子，看看JavaScript代码如何执行核心业务逻辑（用户名和密码验证），然后发送消息（执行函数调用）到服务器（通过HTTP）验证用户认证信息，进而更新DOM，向用户显示更新后的信息。

如果没有集成点，对于业务逻辑而言，DOM和HTTP是什么呢？

业务逻辑通过HTTP调用服务端，这与服务端的代码通过Web Service与其他系统交互的方式如出一辙。客户端肯定与服务端集成，用于集成的代码（也就是Ajax调用的部分）则是我

① http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod
② 要更详细地了解stub和mock之间的差异，请参考Martin Folwer的文章：http://martinfowler.com/articles/mocksArentStubs.html。

们的防腐层[1]（anticorruption layer）。

再说一次，从业务逻辑的观点上看，DOM是集成点。实际上，业务逻辑松散地使用DOM作为数据存储：它从一个节点获取信息，更新节点，修改其属性，以及增删节点。可以说，JavaScript层是通过CRUD操作同DOM进行交互的。

如果把DOM和HTTP看作集成点，实际上就定义了两个清晰的系统边界。我们接下来识别业务逻辑和系统其他部分的分离。从测试的角度看，由于这些边界的存在，我们可以模拟核心业务逻辑的外部依赖，事实上也就是从浏览器（以DOM的形式）和服务端去掉了业务逻辑代码的依赖。现在，我们有了三个清晰的抽象。

- 展现

展现层抽象都是关于如何将应用展现给用户的。举例来说，保证表格中每个交替行都标记了正确的高亮，或者验证消息边上图标正确与否都属于展现问题。一般说来，我们要确保与展现相关的常用功能都在一起，以保证跨应用的一致性，进而确保给用户提供一致的视图。

- HTTP

HTTP抽象用来与服务器交互。最为显著的例证便是必须的Ajax调用。然而，如果应用程序用到了Web socket[2]，也需要将其包含进来。

- 应用逻辑

这是位于应用程序核心的代码。凡是事关应用程序运行的规则都应该当做应用逻辑。这包括了所用的验证规则，以及如何响应用户输入。

大部分标准Web应用很容易套用这样的模型。

前面定义的抽象与著名的被动视图[3]（passive view）模式很相似。我们想强调一下被动视图模式是怎样成为这个问题的合理解决方案的，但不同的应用程序可能有不同的需求，需要不同的抽象。重要的是，确保应用划分成职责不同的区域。

记住这些概念以后，我们现在重构代码，提取出正确的抽象：

JavaScriptTesting/loginPageLogic.js

```
function LoginPageLogic(view, authenticationService) {
    this.init = function() {
        view.addLoginHandler(this.validateCredentials)
    };
```

---

① http://c2.com/cgi/wiki?AnticorruptionLayer

② http://en.wikipedia.org/wiki/WebSockets

③ http://martinfowler.com/eaaDev/PassiveScreen.html

```
    function credentialsAreValid(username, password) {
        return (username && username !== "") && (password && password !== "");
    }

    this.validateCredentials = function() {
        var username = view.getUsername();
        var password = view.getPassword();


        if (credentialsAreValid(username, password)) {
            authenticationService.login(username, password,
                    view.showLoginSuccessful, view.showLoginError);
        }else {
            view.showInvalidCredentialsError();
        }
    }
}
```

现在绑定代码变成了这样：

JavaScriptTesting/loginPageLogic.js

```
$(document).ready(function() {
    var serviceUrl = "http://localhost/authentication";
    var authService = new AuthenticationService(serviceUrl);
    var loginPageView = new LoginPageView();
    var loginPageLogic = new LoginPageLogic(loginPageView, authService);
    loginPageLogic.init();
});
```

现在`LoginPageLogic`清晰地展现出要执行的业务逻辑。所有与服务器和UI的交互分别放到了`AuthenticationService`和`LoginPageView`中。

在很好地隔离了核心逻辑之后，现在可以考虑如何给这部分代码写测试了。特别是，我们可以使用mock程序库编写交互测试，验证`LoginPageLogic`与其协作之间预期的交互确实发生了。

使用JSTestDriver[①]作为单元测试工具，JSMockito[②]作为mock程序库，我们现在可以开始编写测试了，验证`AuthenticationService`没有返回错误的情况下是否调用了正确的回调函数。

JavaScriptTesting/tests/loginPageLogicTests.js

```
test_calls_auth_service_with_correct_callbacks : function() {
    var loginPageViewMock = mock(LoginPageView);
    var authServiceMock = mock(AuthenticationService);
```

① http://code.google.com/p/js-test-driver/
② http://jsmockito.org/

```
        var loginPageLogic = new LoginPageLogic(loginPageViewMock,
                                                authServiceMock);
        loginPageLogic.init();

        when(loginPageViewMock).getUsername().thenReturn("username");
        when(loginPageViewMock).getPassword().thenReturn("password");

        loginPageLogic.validateCredentials();

        verify(authServiceMock).login(is(equalTo("username")),
                                  is(equalTo("password")),
                                  is(equalTo(loginPageViewMock.showLoginSuccessful)),
                                  is(equalTo(loginPageViewMock.showLoginError))
                                );
    }
```

我们还可以编写一个测试，验证在认证校验失败的情况下，是否向用户显示了正确的错误信息。

JavaScriptTesting/tests/loginPageLogicTests.js

```
    test_shows_login_error_if_password_not_entered:function() {
        var loginPageViewMock = mock(LoginPageView);

        var loginPageLogic = new LoginPageLogic(loginPageViewMock, null);
        loginPageLogic.init();

        when(loginPageViewMock).getUsername().thenReturn("username");
        when(loginPageViewMock).getPassword().thenReturn("");

        loginPageLogic.validateCredentials();

        verify(loginPageViewMock).showInvalidCredentialsError();

    }
```

这两个测试明确地测试了业务逻辑，没有依赖服务器端和DOM。这是系统的核心，是真正要验证的部分。通过验证所有的协作者是否被是否正确地调用，以及消息编排按预期运作，就可以做到这一点。

识别出来的协作者又是什么样的呢？

这里识别并引入了两个协作者：

- LoginPageView；
- AuthenticationService。

AuthenticationService的代码如下：

JavaScriptTesting/authenticationService.js

```
function AuthenticationService(serviceUrl) {
    this.login = function(username, password, successCallback, errorCallback) {
        browser.HTTP.post(serviceUrl, {username:username, password:password},
                          successCallback, errorCallback);
    };
}
```

以及

JavaScriptTesting/browser.js

```
browser.HTTP = {
    post : function(url, myData, successCallback, errorCallback) {
         $.ajax({
         url: url,
         type:"POST",
         data:myData,
         success: successCallback,
         error: errorCallback

         });
    }
}
```

这段代码很好理解。为了鉴权认证信息，需要通过Ajax调用来与服务器通信。browser.HTTP就是一个简单的包装器，减少底层调用的繁琐操作。

LoginPageView更有意思些。

JavaScriptTesting/loginPageView.js

```
function LoginPageView() {
    this.getUsername = function() {
        return $("#username").val();
    };

    this.getPassword = function() {
        return $("#password").val();
    };

    this.addLoginHandler= function(callback) {
        $("#loginButton").click(function(e) {
            e.preventDefault();
            callback();
        });
    };

    this.showLoginSuccessful = function() {
        browser.Animations.showMessage("#message", "Welcome back!");
```

```
    };
    this.showInvalidCredentialsError = function() {
        browser.Animations.showError("#message", "Please enter your login details");
    };
    this.showLoginError = function() {
        browser.Animations.showError("#message",
                "We were unable to log you in with the details supplied");
    };
}
```

LoginPageView就是所有UI操作的大门。

LoginPageView的真正价值不在于它的职责（它不应该有任何实际职责），而在于它为其他代码（业务逻辑和测试）增添了语义价值。

看一下测试，就可以看出它们是多么简单易懂，容易遵守，这是因为LoginPageView使用的是视图领域语言，仅仅只描述想做什么，而不是如何做。

没有这一层，业务逻辑和测试将与jQuery直接通信，这样就会降低代码的表现力，继而产生很难遵循和维护的测试。

视图越简洁越好，易于委托给其他底层的协作者。有了这样的简单实现，这个对象就没有必要再测试了，如果视图里出现需要测试的逻辑，就应该视作一种“腐坏”代码（表示逻辑应该移到业务层）。

我们想要在UI上执行的操作复杂度各不相同。有时也许只是一行操作，但更多时候是一组DOM操作，它们只有当做一个整体执行才有意义。

对基本的操作而言，直接与底层的JavaScript栈（也就是jQuery）通信也是可以的；然而，当需要一组操作，并且作为一个整体重复执行时，我们推荐将这些交互隔离到函数里，并使用不同的命名空间分组，比如brwoser.Animations。

JavaScriptTesting/browser.js

```
browser.Animations = {
    showMessage : function(selector, message) {
        $(selector).text(message);
        $(selector).removeClass("error");
        $(selector).fadeIn(2000);
    },
    showError : function(selector, error) {
        $(selector).text(error);
        $(selector).addClass("error");
        $(selector).fadeIn(2000);
    }
}
```

这些函数是用户体验的基础。想象一下，showMessage()被多次重用。例如，如果要将淡出动画从2000毫秒变为3000毫秒，我们可能会漏掉其中的一些，导致终端用户产生不一致的体验，这样最终会降低应用的品质认知度。但通过抽取上述这些函数，不仅遵守了DRY（Don't Repeat Yourself）原则[1]，还可以在底层测试其功能。这种情况下，我们确实希望编写测试验证与DOM的底层交互（也就是与DOM的集成），使用HTML夹具就可以设置和验证这种行为。

JavaScriptTesting/tests/browserDisplayTests.js

```
TestCase("BrowserDisplayTests", {
    test_show_error_displays_error_correctly: function() {
        /*:DOC += <div id="message" class="message"></div> */
        browser.Animations.showError("#message", "error message");
        assertEquals($("#message").text(), "error message");
        assertTrue($("#message").hasClass("error"));
    }
});
```

在这个测试中，一旦定义好夹具[2]，调用待测函数，就可以根据某个特定节点的属性进行判断——验证操作是否成功运用到了DOM上。

6.4　测试方式

这节我们来总结一下客户端JavaScript的测试方式。

6.4.1　倾向于交互测试，而非集成测试

前文介绍的方法显然提倡的是交互测试的风格，而不是集成测试。

在客户端JavaScript中，我们视集成测试为一些需要特定设置的测试，比如DOM可用，或服务器正常运行。如果测试需要HTML夹具才能运行，它就是集成测试（这里是与DOM的集成）。同样，如果测试需要服务器响应Ajax调用，也属于集成测试（它在测试与HTTP和特定服务器的集成）。

有了清晰的关注点分离，在纯粹基于交互的测试风格里，我们可以让测试专门验证应用逻辑是否向其协作者和组件发送了正确的消息。我们相信，客户端JavaScript库规避了不同浏览器DOM实现的差异，所以在系统这一部分进行完整的测试覆盖，不会有足够的回报。最需要测试覆盖的JavaScript关键部分是业务逻辑和表现层、服务层之间的协作，以及业务逻辑本身。

6.4.2　在具体用例中使用HTML夹具编写集成测试

HTML夹具只是另一种形式的代码重复，而且很快就不再与原来的前段标签同步。当然，我

① http://c2.com/cgi/wiki?DontRepeatYourself
② 我们用了JSTestDriver的注释语法声明夹具。

们可以尝试编写非常通用的夹具，让HTML片段更能抵御变化，但这会降低测试的领域特定性，失去了作为一种文档形式的价值。我们相信，编写良好的测试比用其他形式编写的文档都更具表现力，过期（或者太通用）的HTML夹具会导致测试难以表明实际的测试目的。

当然，对有些场景来说，使用集成测试和HTML夹具也相当重要。

如例子中所示，我们想使用夹具验证复杂的UI交互（多个DOM操作实现一个效果）。依赖高级选择器遍历DOM获取具体节点时，我们也希望使用夹具。依照经验，这是典型的跨浏览器问题，我们不止一次在IE 6和IE 7（或更多的现代浏览器）上出现问题。如果拥有可以在所有目标浏览器上运行的测试，可以保证功能与预期一致，也有助于规则退化。

6.4.3　使用验收测试验证所有组件的集成

使用JavaScript时，我们喜欢在单元测试级别工作，但这么做可能无法轻易覆盖事件，判定是否绑定到了正确的HTML组件上。

不过依照我们的经验，这个方面的错误是很少的；给组件添加错误事件几率极低，而且通常在开发阶段就会发现。因此，在这个方面不需要大量测试。就总体的测试策略而言，通常会有一些基于浏览器的验收测试覆盖主要场景。在UI层面，这就是为了自动验证（即便是在很高的级别）大多数处理器是否能正常工作。



小乔爱问

我能TDD JavaScript吗？

当然可以，我们可以TDD JavaScript代码。

只要不再把JavaScript认为只是一种用来钻研技巧的语言，我们熟知的设计和编码技术就都可用。

6.5　持续集成

有了测试，我们当然希望把它作为构建的一部分，在每次检出（check in）代码时提供最好的反馈。JavaScript的测试运行得很快，上百个测试仅需几秒就可完成。

6.6　工具

本文写作时，JavaScript工具家族中依然有许多积极的演进尚在进行，（还）没有哪一种工具成为必然的标准。对我们而言，关键的决定因素是每个选定的工具应该能轻松地与持续集成服务器（Continuous Integration Server）集成。

对于测试而言，我们推荐使用真正的JavaScript引擎，理想情况下，测试可以在多个浏览器上运行。如上所述，我们推荐交互测试，而非集成测试。但我们相信，对测试策略而言，在目标浏览器上使用HTML夹具运行集成测试是很关键的。

6.6.1　单元测试

JSTestDriver[①]是一个语法简明而又整洁的工具，可以同持续集成服务器很好地集成，配置简单而强大，能够并行地在不同的浏览器上执行测试。

6.6.2　语法检查

另一个重要工具是语法检查器。目前我们最喜欢的工具是JavaScriptLint[②]。它简单易用，可配置度高，可以按照需要进行检查。

检查时至少要看一看代码中是否有分号缺失（最小化[③]（minification）和压缩可能会因此无法正常工作），但是，大部分工具都是高度可配置的，可以确定哪些东西必须检查。语法检查非常重要（且迅速），因此我们把它放在构建流水线的第一步执行。如果代码无法通过初始检查，就没有必要启动其他代价更高昂的任务，比如编译（是的，编译完服务器端代码后才运行它），或功能及验收测试。

6.6.3　mock框架

交互测试很重要，而有一个好的mock程序库也是关键所在。我们需要一个工具，以便能够定义函数调用的期望，以及定义这些函数调用的返回值。JSMocktio是我们最喜欢的工具之一。它受到Mockito[④]的启发，是一个著名的Java mock程序库，语法简单，还能与JSHamcrest[⑤]匹配器很好的集成，让断言可读性非常高。

6.7　总结

HTML5的出现和对富互联网应用程序的关注，会给互联网应用程序类型带来极大的转变。只有确保我们编写的代码能够演进，才能保证在满足业务需求的同时，提供最合理的用户体验。JavaScript作为第一类语言，其关注度也日渐增加，而且富互联网应用程序也在增加，作为开发者，我们写出的JavaScript代码不仅要简洁，还要可读、可测、可维护。因此，只有应用良好的设计实践，才能确保今天编写的应用程序可以应对未来的挑战。

① http://code.google.com/p/js-test-driver/
② http://www.javascriptlint.com
③ http://en.wikipedia.org/wiki/Minification_(programming)
④ http://mockito.org/
⑤ http://jshamcrest.destaquenet.com/

构建更好的验收测试

James Bull撰文

编写自动化验收测试套件时，经常会面临很多问题；验收测试可能会变得很慢很脆弱，而且不可维护。希望在本文结束时，你可以掌握如何构建快速、有弹性且可维护的验收测试。此外我们还会介绍，验收测试如何适用于软件开发过程，以及它将如何影响本文提及的优秀实践的采纳。文中的实例都来自于网站测试，但其背后的理念却是广泛适用的。

我们先从定义开始。验收测试应该符合以下所有条件：

- 通过用户界面驱动；
- 运行于整个软件栈之上；
- 尽量覆盖真正的集成点；
- 完全自动化；
- 作为持续集成构建的一部分运行。

验收测试的目标是，增强随时都能发布适当产品的信心，大幅度减少手工执行回归测试的时间。

7.1　快速测试

快速测试能更频繁地获得关于软件质量的反馈。运行测试构建得越多，效果越好。为一个特定变化获得反馈的时间越长，bug在软件中隐藏的时间就会越长。我们很难定量说明什么叫“太慢”，但是，如果每天不能多次运行测试或有用的子集，那么给测试加速就会带来极大的好处。

7.1.1　基于用户行程的测试

因为验收测试运行得比单元测试慢，所以我们会发现验收测试难以面面俱到，而且运行速度也不是很快。自动化验收测试应该只是整个自动化测试策略的一部分。在验收测试中所做的断言，在单元测试层面中也覆盖到了；通过用户界面进行测试时，我们并不准备测试所有的极端情况，

而只是看看所有应用层是否能够协同工作，成为一个协调运行的整体。

应该把测试定位成覆盖最常用的功能。找出系统中最重要的部分，尝试测试贯穿这些部分的主要路径。要做到这一点，应该考虑一下用户使用系统以达成系统主要目标的用户行程，这是一种行之有效的方式。

首先找出几个人物角色，分别代表不同类型的典型用户，想象一下他们会如何使用网站。找出从哪开始，思考网站的业务目标，并聚焦于此。

以ThoughtWorks的网站为例，我们识别出的人物角色可能包括如下两位。

- 有好奇心的开发者Dave：他对ThoughtWorks的工作流程较感兴趣。
- 潜在客户Clive：他对ThoughtWorks此前提供的客户体验较感兴趣。

为每个人物角色创建一条贯穿系统的用户行程，该行程要能够代表上述两位用户。测试要能够准确地反映例子里用户会做的事情。在这个例子里，我们会让Dave访问网站的职业生涯板块，而Clive则会选择浏览客户体验报告。

这不同于传统的方式，因为它会在单个测试中覆盖到系统中的很多功能。当然，这种方式也有缺点，通常我们会通过将测试拆分成小块将其避免。但在该测试中我们会发现，如果测试失败，找出问题代码却并非易事，因为有很多问题都可能导致失败，继而造成测试失败。然而这也不算大问题，这里覆盖的所有内容都都会覆盖相同代码的单元测试，而且愈到底层愈加详细。

这种方式会带来速度上的优势，它不像那些只关注系统某一部分的测试，多次访问同一个页面，我们只会访问每个页面一次，按照行程遍历系统，这样就大幅降低了执行时间。

7.1.2　并行执行测试集

要做到这一点，我们可以使用像Selenium Grid这样的工具。然而，这么做会把我们限制在某个特定的工具上，因为它只能与Selenium测试协同工作。如果我们已经有了Selenium测试集，只是想加速而已，那这种方式就是最好的选择。

如果使用不同的工具，那么让运行测试子集更容易不失为更好的办法。采用这种方式，我们可以在多台机器上检出代码，在不同的机器上运行测试集不同的部分，以此并行。也可以利用大多数持续集成软件做到这一点，只要花费很小的代价，就可以得到一份全面的测试报告。但如果这么做不可行，那就有点麻烦了，要手工部署不同的配置文件到不同的机器上，让它们把报告写到一个公共的文件共享里去。不过切分测试集执行不需要写太多的代码。我曾经使用命名规范，让运行不同的测试变得简单。如果看一下示例，就会发现我把测试分别命名为`DeveloperDaveUserJourneys`和`ClientCliveUserJourneys`。这样，使用Ant的jUnit任务，就可以运行特定用户的测试，或是所有的用户行程了。

BuildingBetterAcceptanceTests/AnthologyAcceptanceTests/build.xml

```
<target name="test" depends="compile">
    <junit printsummary="yes" haltonfailure="no"
           showoutput="true" fork="yes" forkmode="perBatch">
        <jvmarg value="-Dweb.driver=${driver.type}"/>
        <classpath refid="classpath"/>
        <formatter type="plain" usefile="false" />
        <batchtest>
            <fileset dir="${test.dir}">
                <include name="**/*${tests.to.run}*.java" />
            </fileset>
        </batchtest>
    </junit>
</target>
```

7.1.3 考虑使用多种测试驱动器

还可以想一想如何给浏览器驱动器接口提供多个实现。之所以提及这一点，我想说的是不要直接使用工具编写测试，而是声明一个接口，以此编写测试，然后提供一个适配器，处理对工具的调用。这样，我们的测试集就会有多个驱动器的实现了。

如果有一个驱动浏览器的实现，一个无法驱动浏览器的实现。那么通过浏览器运行测试可靠性高，而以非基于GUI的驱动运行，则可以提升速度。这样我们就可以在提交之前运行更多的测试，而作为CI的一部分，我们仍然可以通过浏览器运行全部的测试集。

7

WebDriver已经这么做了。它有多个实现，包括HTMLUnit、Chrome、Internet Explorer以及Firefox。下面的例子用到了WebDriver，我用了一个叫`ApplicationTestEnvironment`的静态类获取所有需要测试的页面，并注入适用的驱动器。驱动器的类型是通过命令行传给测试的，它保存为JVM的一个属性，我们在前面例子已经看见过了。

有两种方法：

BuildingBetterAcceptanceTests/AnthologyAcceptanceTests/src/Utilities/ ApplicationTestEnvironment.java

```
public static Object getPage(Class c){
    try {
        Page p = (Page) c.newInstance();
        p.setDriver(getDriver());
        return p;
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
    return null;
}
```

如果测试要获取一个与之交互的页面，就会调用这个方法。它会调用`getDriver()`得到一个合适的WebDriver实现。

BuildingBetterAcceptanceTests/AnthologyAcceptanceTests/src/Utilities/ApplicationTestEnvironment.java

```
private static WebDriver getDriver(){
    String driverType = System.getProperty("web.driver");
    if(driverType.equals("browser")){
        if(driver==null){
            driver=new FirefoxDriver();
        }
        return driver;
    }
    else{
        HtmlUnitDriver newDriver =  new HtmlUnitDriver();
        //newDriver.setJavascriptEnabled(true);
        return newDriver;
    }
}
```

这样，我们就可以通过命令行轻松地指定要执行的测试和要用的驱动器了。只要一个很简单的批处理文件，比如下面这个：

BuildingBetterAcceptanceTests/AnthologyAcceptanceTests/Test.bat

```
ant -Ddriver.type=%1 -Dtests.to.run=%2 test
```

这样一来，我们就可以在命令行里运行测试。下面有三个例子，第一个通过浏览器运行一个用户，第二个表示用无头（headless）的HTMLUnit驱动器运行，第三个则是用浏览器。

```
Test browser Dave
Test headless Clive
Test browser UserJourneys
```

这种设置让我们可以轻松地选择使用哪个驱动器，以不同的方式运行同一测试。

这种做法也有潜在的缺陷。如果网站大量使用JavaScript，并且无法在禁用JavaScript的情况下工作，那么使用HTMLUnit会遇到些麻烦，因为HTMLUnit使用的JavaScript引擎不同于普通的浏览器。WebDriver默认会禁用HTMLUnit的JavaScript，所以如果必须要用JavaScript，最好还是用完整的浏览器驱动器。如果用完整的浏览器驱动器，就可以用到特有的一些功能，比如鼠标悬停（hover）。

我曾经参与过一个项目，那个网站大量使用了JavaScript，但我们依然成功地使用了无头驱动器。我们当时有个特定的需求——网站在禁用JavaScript的环境下依然能够正常工作，而在有JavaScript的情况下，能够渐进增强。这样的需求和上面提到的测试策略结合起来，让那个项目得以成功。

7.1.4　将测试分开运行

如果已经对测试集进行了有效地组织，使其并行运行，还用上了无头驱动器，但还想加速测试，那么请考虑让测试分开运行吧。

划分出一组高风险的测试，将其作为主要构建的一部分运行，并把这个子集当做常规开发活动中的活跃测试集。剩下的测试可以在一个单独的“缓慢构建”流水线上并行运行，抑或是在受限的构建环境中夜间运行。如果缓慢测试中的一个测试失败了，那就把它挪到活跃集中。如果活跃集中的某个测试很长时间都没有失败，那就考虑把它挪到低风险测试集中。

通过之前用过的命名规范就可以实现上述操作。但如果需要更好地控制运行哪个测试，那我们可以花些额外的功夫，把所有测试单独放到配置文件中引用，然后写些代码检查最近的*x*个构建结果，并按需改写配置。

7.1.5　等待页面元素显示时要小心

如果要测试的页面中含有Ajax请求，而且在继续之前要等待它返回，那么我们可能会尝试休眠一段时间，等待页面元素出现。然而，如果我们这么做了，会降低测试集执行的速度，尤其多个测试都执行这一操作的情况。相反，我们可以进入一个带有超时的循环，重复检查需要的元素，直至它出现，然后点击它。

循环越紧，测试运行越快。唯一的例外可能是根本不休眠，但那样会占满CPU，实际上会更慢。如果用WebDriver尝试这种方式，请记住，WebDriver是通过网络与浏览器通信的，所以如果重复轮询某个东西，就会耗尽Socket。我认为原因如下：为了加速构建，减少了等待循环的休眠时间，然后测试就会定期失败，而且看起来毫无规律可循。

为了让测试运行得快，又不致于耗尽Socket，我们为休眠间隔实现了一个“让步策略”（back-off strategy），第一次轮询之后等待10毫秒，第二次等待20毫秒，以此类推，每次加倍，直至最大值为2秒。这个策略很好用，测试会运行得很快，而且还很稳定。

7.2　有弹性的测试

有弹性的测试（resilient test）就是不该失败时就不会失败的测试。我曾见过有的项目付出很多精力创建验收测试，随后又将其弃之脑后，因为测试总是谎报军情，让人们无法相信测试结果。这种代码没问题也会失败的测试叫做脆弱的测试（brittle test）。

脆弱的测试没什么用。想想狼来了的寓言吧。如果测试在代码没问题的时候总是提示失败，很快开发者就会忽略掉这个问题，被忽略掉的测试便毫无价值。

更糟的是，如果代码没问题，但却出现测试失败，这会给我们带来额外的负担，因为我们需要想办法修复这种测试。

所以，有弹性的测试的价值就体现在其值得信任和易于维护上。

7.2.1　单独选择页面元素

这里我要说的是，我们应该能够持有页面上的每个元素，而无需引用其他元素。有些工具可以通过XPath选择元素，实际上我们可以自行设置工具，找到想要的元素。

找到左边的第三个div，右边的第二个ul，三步之后就能得到想要的按钮。

看看下面这个例子，它来自一个真实的网站，采用Firefox的Selenium插件录制：

```
"//div[@id='show']/div[2]/div[7]/ul/li[2]/a/span[1]"
```

如果重新布局页面，比如把回退按钮（back button）放在清除按钮（clear button）的另一边，那么路线发生变化，测试就会失败。如果能提供一个地址，让工具不管实际位置都能找到元素，则不失为一个好办法。

不用XPath还有一个原因，即不同的浏览器实现XPath略有不同，因此，XPath在IE下选择的元素可能不同于Firefox。这是因为IE的计数从0开始，而不是W3C指定的1。这让跨浏览器测试集更加难以实现。

幸运的是，支持XPath的工具通常也支持其他可以选中页面元素的方式。一般说来，要想让一个页面易于测试，我们需要为每一个要与之交互的元素提供一个唯一标识符。设置ID相当方便，而且它必须是唯一的，因而能够很轻松地加在每个与之交互的元素上。

但处理列表时则有点麻烦，因为ID显然是不可以重复的。

在最近的一个项目中，我们就在测试里处理过列表，只要把每个列表用div包起来，就可以唯一地识别列表。随后，应用为列表中的每个元素生成ID，以包装列表的div的ID为基础，在每个列表项上添加一个数字。

如果无需依赖于其它元素的位置，也能够找到其他方法唯一地定位页面元素，倒也不必拘泥于使用ID做标识符。

比方说，如果CSS结构很好，且元素的class不太可能发生变化，就可以使用列表的ID和`css`类获取列表元素。

如果只选择元素，可以规避测试和页面布局的绑定问题。这样，页面修改或样式改变时，测试如果失败，唯一的原因只会是功能本身发生了变化。

7.2.2　等待页面元素显示时要小心（再次强调）

等待页面元素出现时的方式不仅会影响测试的性能，也会影响测试的弹性。

与大多数Web测试工具一样，WebDriver通常会等待页面加载完毕。但如果页面含有异步Ajax请求，工具便无法得知页面何时才能加载完毕，我们只好自己编写代码来实现了。

在7.1节提及的轮询是一种很好的机制，因为除了等待固定时间之外什么都不做，会让测试脆弱或缓慢。我觉得这里再次提及这一点非常重要，因为根据我的经验，有很多虚假的测试失败都是由于没能等到正确的元素造成的。

下面这个例子可以说明其中的困难之处。想象这样一个应用程序，它会用整数度数显示当前的气温；它不会自动更新，但有一个刷新按钮，可以发送Ajax请求，取得气温的最新值。

如果我们点击按钮，气温显示却没有改变，我们怎么才知道它探测到了气温，但气温保持不变；还是应用根本就没起作用，没有更新温度呢？

即便人工识别，如果不改变传感器的温度，都很难判断这样一个系统是否正常工作，那怎么才能在自动化测试里判断何时停止等待呢？我们要等的不是气温这个元素，因为它已经在那了，也不是要等它的值改变，因为即便应用程序没有差错，气温也可能保持不变。

唯一知道温度是否发生变化的方式，是用一种方法来允许我们修改读取温度的代码，这样就能模拟温度的变化了。

其他场景中也存在同样的问题，比如每个页面都有固定的导航条，用作站内导航。以此为例，我们假设所有的页面都有后退的链接。

点击后退按钮，我们该等待什么呢？不能等后退按钮显示，因为它早就显示出来了，这样一来，无论页面是否改变都会立即返回；也不能等页面模型中某个元素的出现，因为我们根本不知道上个页面是哪个页面。唯一能做的，就是在页面类中写一个等待的方法，让测试在适当的时机调用该方法。

但这种方式很容易出错，而且每个系统的易错点也各不相同。然而，我们还是能做一些事情，让我们在这方面不至于太累。首先，找出哪些页面含有Ajax调用。没有Ajax调用的地方不需要任何等待，驱动器会自动等待页面刷新完成；有Ajax调用的地方，确保我们没有每次重复实现等待功能，并且自己写几个工具方法。

```
waitUntilPresent(id)
waitUntilGone(id)
```

在页面模型类里使用上述方法，给测试暴露一些良好命名的方法，这样使用哪个方法就显而易见了。如下所示：

```
page.waitForCheeseSelectorToDisappear();
page.waitForBiscuitWidgetToBeVisible();
```

如果能够仔细思考怎样在测试里等待元素出现，我们就会对测试的弹性更有信心，也不会有时断时续的超时问题了。

7.2.3　在测试中设置测试依赖的数据

这是一种不错的方法，因为它能保证测试所需的数据在测试运行时总是可用的，这样我们就可以在本地或测试环境中随意摆弄测试数据，也无需担心会破坏测试。这样，我们也就无需手工设置或管理测试数据了。实现这一点的最佳方式是，使用应用程序本身的数据管理代码。

测试启动前，先清空数据库，插入所需数据，然后运行测试。

这么做的原因在于，如果我们维护单独的数据设置机制，比如SQL脚本，那么数据模型改变时，相关的数据库改变会破坏测试。这样一来，我们发现数据模型的任何修改都要做两次：一次是应用，一次是测试。只要老的功能保持不变，我们其实不关心数据持久化的方式是怎样支持新功能的。如果测试依旧通过，系统就还拥有所需的数据，还在尽着它的职责。

怎样利用应用的数据管理代码取决于个人。你也许把测试放在同一个代码库里，在这种情况下，系统中直接包含所需的代码；如果把测试放在不同的代码库里，或是用另外一种语言编写，就要花些力气给测试提供一个接口。如果我们做的是网站，最简单的方式就是提供一个Web Service，这样就可以通过工具进行HTTP调用设置数据。但这个Web Service只能调用代码库中已有的数据入口方法。

也许在我们的应用里，系统会用到大量的数据，但却无需修改。在这种情况下，我们只要维护一个单独的测试数据库即可，升级方式等同于产品数据库，这样还会覆盖部分数据库的升级过程。我见过很多项目都是用DBDeply做这件事的，而且做得还不错。因此，我们只要编写一个数据库清理器，把系统弄回已知的状态就好，而且不必每个测试都清理干净，从头创建所有的东西。如果我们发现应用数据层并不支持要设置的测试数据，那么就应该在应用代码库里自行添加，使其支持要设置的测试数据。

使用应用程序的数据模型降低了需要修改的代码数量，这意味着测试的数据设置与应用程序的其他部分都依赖同一个代码库。这样我们编写的代码就会更少。数据访问代码里的bug也很容易暴露，因为测试一旦没有得到预期的数据，就会失败。更少的代码意味着更低的成本和更有弹性的测试，因为我们已经避免了数据的测试视图与应用程序视图之间的不同步问题。

7.2.4　测试集成点

我说的集成点指的是，系统需要在正常操作中用到的既有系统。举例来说，作为每次页面加

载的一部分，我们都要调用外部系统进行点击追踪。另一个例子是，有一个服务，可以提供我们所依赖的系统用户的信息，并对用户进行认证。如果外部系统不能正常工作，或是我们不能以正确的方式与其通信，那么编写的软件也就无法按预期正常工作。

很明显，如果我们依赖真正的集成点，就可能在自身没有错误的情况下导致构建失败。尽管这种情况非常恼人，但它提供给我们的信息却是非常重要的，因为开发过程中遇到的外部系统的问题，在产品环境中也很可能遇到。

我们想做的是，在集成点出问题时，确保不会花大量时间在自己的系统中找寻bug。要做到这点最好的做法是，在验收测试之前，构建一个专门测试集成点的部分。因为验收测试要比单元测试触及更多的系统部分，准确找出失败也要更困难一些。通过测试集成点正常工作与否，能够返回我们期望的信息，也就不难确定是不是第三方出了问题。这么做还有一点额外的好处，即在代码未被破坏的情况下，又杜绝了一种验收测试失败的可能性。

理想情况下，我们会测试集成点，然后依赖真正的集成点，继续运行验收测试。但有时，这是不可能做到的。也许与之交互的系统根本没构建，也可能尚在开发之中，我们会发现构建经常会失败。这种情况下，我们只好用桩（stub）代替真实的集成点。我们应该运行所有的集成测试，因为这至少可以保证，按照预想的最终集成点运作方式，我们的代码可以正常运作；而当真正的集成点可用时，就可以用真正的集成点运行测试了。

我们的目标是尽量减少测试失败的可能性。理想情况下，引起测试失败的原因只有一个，就是我们要测试的东西真的破坏了。通过ID进行选择，我们移除了因为布局变化造成的失败；通过复用应用的数据访问层，在每个测试里设置和删除数据，我们移除了因为数据库变化造成的错误；通过分离集成测试，我们消除了其他东西破坏构建的可能性。

有弹性的测试相当不错：维护简单，值得信赖，而且可以用来作为衡量软件质量的指标，因为它们只会在有东西真正出问题时才会失败。

7.3　易于维护的测试

让测试具有弹性只是写出易于维护的测试的方式之一。

7.3.1　使用页面模型

如果有几个测试访问同一个页面，但却做不同的事情，我们可以在每个测试里直接使用页面的URL。但如果这么做，当URL发生改变时，就不得不修改所有用到它的测试。相反，把测试里的页面细节放到一个类里，就可以重用细节。这样做之后，当页面细节发生改变时，我们只需在类中修改一次即可。

page类背后的目的是，同页面的交互应该提供一种符合直觉并清晰简洁的描述，体现出浏览器驱动器实际做的事情。下面是从一个样例中抽出的代码片段，能够体现我要表达的意思：

BuildingBetterAcceptanceTests/AnthologyAcceptanceTests/src/tests/CuriousDeveloperDave/DeveloperDaveUserJourneys.java

```
TWHomePage homePage =
    (TWHomePage) ApplicationTestEnvironment.getPage(TWHomePage.class);
homePage.visit();

Assert.assertTrue(homePage.Menu().exists());
Assert.assertTrue("Careers link not present",
                  homePage.Menu().CareersLink().exists());

Menu menu = homePage.Menu();

CareersPage careersInfoPage = menu.clickCareersHomeLink();
Assert.assertEquals("href for careers section on menu not correct url",
                    menu.CareersLink().href(), careersInfoPage.url());
```

我们从应用测试环境中得到第一个页面，然后，所有进一步的交互都是通过这个页面完成的。我们访问这个页面，做了一些断言，然后点击了该页面上的一个链接。在这个例子里，我们还有一个menu类，因为ThoughtWorks网站上的所有页面共享着同样的菜单，把它单独抽出当做单独的类也是很合理的。

page类本身有一个字段，该字段是实现了WebDriver接口的某个东西的实例。page类了解页面细节，使用驱动器与浏览器进行交互。

BuildingBetterAcceptanceTests/AnthologyAcceptanceTests/src/pages/TWHomePage.java

```
public void visit() {
    driver.navigate().to(url);
}

public boolean FeatureBannerIsPresent() {
    List l = driver.findElements(By.id("banner-display"));
    return l.size()>0;
}
```

使用页面模型让测试更易于维护，因为当有页面发生改变时，只需要修改一处代码即可，无需修改每个用到它的测试。

7.3.2　结构一致的测试集

谈及快速测试时，我提到了用户行程的概念。这个概念还让测试集有了结构（structure）。该结构会使得在某个特定功能区查找测试更容易一些，也更容易知道在哪添加新测试。设想一个人物角色表示系统的真实用户，可以帮我们确定什么是最重要的，帮我们记住每个用户行程。

从样例代码中可以看到，测试集的基本结构是页面、测试和工具。工具包括等待代码和应用测试环境。页面目录包括所有与站点封装元素相关的代码，比如菜单、按钮等。很明显，所有的测试应该放在测试目录下，每个任务角色都应该有自己的包。

维持测试集的一致结构，根据用户人物角色测试，可以减少大量花在维护上的时间，因为我们不必花时间在测试集内苦苦寻觅，找寻特定的页面该在哪里测试，或是费心思考将新测试放在哪里。

7.3.3 测试代码产品代码一视同仁

测试代码支持我们的应用程序。如果我们信心满满，相信自己的测试是一个质量上佳的晴雨表，那么团队就可以大幅度重构，而不必担心引入bug。好的测试让团队极少关注手工回归测试，而更多关注正在开发的最新版本测试，以确保代码从头开始就是正确的。如果我们只关注开发速度而不注意测试代码，测试将会变得脆弱而缓慢，花在维护测试集上的时间会更多，实现的价值自然就会减少。

正因为这样的原因，对待测试代码的严格程度应该与上线代码一样。在合适的时机重构测试代码，而且请铭记，编写测试和快速的构建反馈会让我们的工作更轻松。

在我最近做过的几个项目中，我们先用页面模型编写验收测试，用以驱动出测试所需的ID。我们编写测试，确保它失败，然后实现代码使其通过。当遇到某些需要为边界条件写测试的代码时，我们会立即跳到单元层面测试边界条件，然后再回到验收测试层面。通过这种方式，我们可以同时完成编码和测试，确保它们各司其职。

7.3.4 切勿受限于工具

在我曾经做过的一个项目中就出现过这样的问题，我们的工具用得很表面，都是直接在测试里用。使用另一个工具重写所有测试花了很大的力气，所以测试都被忽略了，最终弃之一旁。维护一套测试集的时间越长，它就会变得越大，其中倾注的时间和精力也越多。随着时间的推移，因为给产品添加新特性，或是我们所用的工具版本过时，修改测试集的可能性也随之上升。

面对这种情况，我们希望测试工具越容易更换越好。要做到这一点，最好的做法是让测试依赖接口，而非具体实现。这样，当我们想做工具迁移时，只要实现一个适配器，适配测试依赖的接口和新工具提供的方法即可。虽然这样做仍然需要付出一定的工作量，但比从头重写所有测试要少得多。

本例中，我们用的是WebDriver。这是个很好的选择，因为它是开源的，而且从代码中可以看出，WebDriver是一个接口，有不同的驱动器实现它：如HtmlUnitDriver、FirefoxDriver，等等。

我认为WebDriver的接口直观易用，我们可以下载源码，用它编写测试。如果未来的某个时候，我们觉得WebDriver本身提供的实现不能满足需求，还可以实现一个适配器，封装要用的测试工具。

使用页面模型，当程序改变时，也不需要对测试做很多修改。给测试集构建一致的结构，利用好用户行程，这会更容易在测试集里找到一个测试，也会更容易确定新测试放在何处。测试代码应与产品代码一视同仁，编写测试代码多花的那点儿时间，会在未来节省出更多时间。使用浏览器接口，则会让工具迁移更加轻松。如此一来，在测试上所花的时间会变少，我们就可以把更多的精力投入到实际的产品开发之中了。

7.4　付诸实践

我们已经讲了好几种可以使测试变得快速、有弹性且易于维护的实践了。我给出的许多建议都需要开发人员紧密地参与到编写测试的过程中去。我建议在页面模型中使用ID。如果不先写测试后写代码，就很难做到这一点，因为如果不先写这种利用ID标记的测试，根本就不知道该在哪个元素上添加ID。如果不参与开发，使用数据层就会很困难，因为正确的方法可能还不可用，抑或编写测试的人不知道该怎么做或怎样引用正确的代码。

其他建议，比如为用户创建人物角色，找出最重要的用户行程，都是业务分析师和QA可以给予重点关注的地方。

大多数的团队都具有实施这些建议的能力，但为什么实施验收测试的尝试时常会走错方向呢？我认为问题在于，任务通常要么交给一组开发人员，要么交给一组测试人员。没有开发人员的帮助，测试人员很难写出易于维护的测试，同样开发人员也不太可能组织好测试集，或是从终端用户的角度考量网站。只有把两组人放在一起，同时编写测试代码和产品代码，才能在正确的时机利用到每个人的力量。

我相信一个能鼓励人们协作的流程至关重要。下面的一些实践源于我们的切身经验，让我们得以用之前描述的方式编写测试。鉴于篇幅有限，不能一一尽述，它们适用于整个开发流程，但希望我至少可以列出它们对编写良好的验收测试所作的贡献。

7.4.1　一地团队

团队应该围绕项目构建，开发人员、测试人员、业务分析师和项目经理都应该坐在同一张桌子旁。团队成员之间的距离应该近到无需走过去或者高声说话，就可以互相问问题的程度。这意味着，当有人有问题时，只管问就好了，他也可以立刻得到答复，而不是找到bug，在某些软件上来来回回拉锯。这有助于交流。

7.4.2　维护测试，人人有责

确保测试具有良好的用户行程，覆盖了系统最重要的部分，这是QA的职责。确保测试集编写良好，易于维护，这是开发人员的职责。确保测试能顺利运行，这是所有人的职责。此外，大家的职责还包括了解测试状态，如果有人破坏测试，他们的职责就是修复应用程序；如果应用程序没问题，那他们的职责就是修复测试，并且，弄清楚为什么应用程序无恙而测试失败。

7.4.3　故事启动

当开发人员准备开始开发新功能时，QA、开发人员和业务分析师应该聚在一起，讨论新功能及其验收标准。该讨论应有如下切实的要求：

- 识别合适的用户行程测试这个功能，考虑测试如何扩展；
- 识别验收标准，用以对已完成的故事进行评估；
- 识别需要进一步分析的不确定部分。

这样做的好处在于：在开发之前，我们就澄清了故事中的不确定性，确保业务分析师的意图同开发人员以及QA的理解是一致的。这样有助于减少bug出现的几率，也降低了对写下来的需求解读错误的几率。通过一起识别验收标准，识别要修改的用户行程，三方共同完成基础工作，确保验收测试合理且值得。

7.4.4　结对测试开发

当开发人员开始做开发时（理想情况下，应该在故事启动之后立即开始），应该和QA坐在一起，写出一个会失败的验收测试。

这样做的好处在于，程序员对于测试集结构的贡献有助于维护测试集，并且确保测试不会是脆弱的。QA的工作则保证了所有达成一致的标准由自动化测试覆盖。此外，双方可以彼此分享知识，如果测试通过，整个团队都会很满意，这一点其实很重要，它能提升我们对于测试的信心。

为本文编写样例测试时，我也遇到了几个问题，但如果身边有开发人员和我一起写测试，这些问题就很容易解决了。

我遇到的第一个问题是，有一部分我想与之交互的元素没有ID。这意味着，我们会基于这一点来做大量的测试，测试还必须用页面元素的相对路径引用元素，这会让测试变得很脆弱。如果我同开发人员一起工作，写出一套唯一的ID就会轻而易举，之后把它们添加到页面上，验收测试就会更具弹性。

我遇到的第二个问题是`menu`类。我的实现方式如下：实例化一个合适的`page`类实例，手工导航到正确的URL。这样做和实际点击页面链接的效果是一样的，但之所以可以这样，是因为我

们知道正确的URL，所以无法再测试链接中的实际URL正确与否。

之所以出现这种情况，是因为我想让这些测试在无头的情况下运行，HTMLUnit驱动器并不支持鼠标悬停，但真实的浏览器却是支持的。WebDriver不能与隐藏元素交互，因此我们需要触发正确的事件，让菜单在WebDriver与之交互前可见。为了正确地测试它，有两种解决方案：第一种是放弃无头运行测试的能力，只用真实浏览器驱动器。如果我们采用这种方案，理想的实现是给要点击的链接一个ID。这会把我们的元素转型成一个可渲染的Web元素，暴露隐藏的菜单，在页面类里与之交互，方式如下：

```
WebElement careersMenu = driver.findElement(By.Id("mainMenu"));
    RenderedWebElement renderableCareersMenu = (RenderedWebElement) careersMenu;
    renderableCareersMenu.hover();
    driver.findElement(By.Id("OurProcess")).click();
```

第二种解决方案是，在JavaScript代码里添加一个钩子方法，让测试变得更容易。这个钩子会触发合适的事件让菜单显示出来。我们需要在HTMLUnit驱动器里启用JavaScript，把它转成JavaScript执行器的做法如下所示：

```
JavascriptExecutor js = (JavascriptExecutor) driver;
js.executeScript("javascriptMethodCall");
```

7.4.5　故事展示

开发完成之后，需要把完成了的特性和验收测试展示给业务分析师以及QA团队。所有的验收标准都应该覆盖到，QA团队会发现之前与开发人员合作编写的失败测试，现在已经可以通过了。

这么做的好处在于，我们已经覆盖了所有显而易见的场景，还拥有了自动化验收测试。现在就可以将故事移交给QA团队，而且无需进一步修改就能通过测试。

7.5　总结

我认为要让验收测试正确工作，最重要的事情是流程。如果不同时写代码和测试，便无法遵循本文推荐的大多数实践。如果不在一起工作，不经常以结构化的方式进行沟通，比如故事启动和故事展示，便无法在正确的时间从正确的人那里获得正确的输入。

如果按这种方式工作，遵循本文的技术建议也会更容易。这样，我们会得到一套运行快速、不易失败、极具价值且易于维护的验收测试。

最终，这些测试会为我们的代码提供一张附加的安全网，以更低的风险更频繁地发布产品。

Part 3 第三部分

软件开发问题

从现代 Java Web 应用到在业务阶段驱动业务创新，4 位 ThoughtWorks 员工分享了各种软件开发中的问题。

第 8 章 现代Java Web应用

Sam Newman撰文

绝大多数Java Web应用都在因循HTTP Servlet API及其容器所设下的藩篱。多年以来，可扩展性都是通过添加更强大的硬件来实现的，这些机器上都运行着多个线程，它们组成集群提供弹性，增强负载能力。许多商用容器的功能列表越变越臃肿，而其厂商却在以此作为卖点反复吹嘘。在Web开发领域，Java Web应用是不被看好的。PHP、Python以及Ruby都在Web开发领域颇有建树，而标准的Java Web应用却对此置若罔闻，闭门造车。愤世嫉俗的程序员认为是软件生产厂商限制了大家看待Java Web应用的思路，而在企业开发环境中，功能越多越有助于厂商销售其产品，而具有简洁性（或是适用性的）的产品反而不受欢迎。

在开发过多个大型Web应用项目后，我们在ThoughtWorks内部形成了一套常识性的方法，这些方法适用于构建高度可扩展、高度可测试的Web应用。我们从其他平台上借用了不少好主意，也希望和Java开发社区的诸位分享这些好方法。这些想法内容如下：

- 无状态应用服务器；
- 按新鲜程度分区及渐进增强；
- 容器外测试和无容器Web应用；
- POST重定向到GET。

值得注意的是，虽然本章主要讨论的是Java Web应用，但其背后的模式同样适用于其他语言（或者是在其他语言中已经有很广泛的应用了）。

8.1 过去的状况

在谈如何编写Java Web应用之前，我们先回顾一下传统的、标准的Java Web应用开发是什么样的吧。

8.1.1 有状态的服务器

HTTP Servlet API给了程序员一种非常方便的方式，可以将数据存储到一个用户session（会话）里，也就是是`HttpSession`。我们可以用这个类存储信息跟踪ID、购物车、历史记录，等等。从程序员的角度来讲，抽象出一个session类可以带来很多好处。然而，这样做有两个明显缺陷：故障转移和性能。

只要用户需要，session就必须保留在内存中。但是到底存多久呢？如果用户注销，我们可以清理内存，但大多数用户并不会显式注销，而且有些情况下，我们甚至还得在用户未登录的情况下跟踪session。但如果这样，服务器端怎么知道何时该清理session呢？服务器端至多只能在用户一段时间没有动作之后清理session，除此之外别无办法。虽然我们还可以通过调整session超时的时间长度减少内存消耗，但如果session保存的时间太短，用户会因为状态丢失而心生不满；时间过长，则面临着占用内存问题。

在内部实现里，Servlet容器是通过传递cookie识别用户的。session数据并没有通过网络传递；相反，它通过cookie确定读取哪份session数据。对于许多使用这种机制的人来说，在后端有多个Web服务器，前端有一个负载均衡器时，往往就会遇到麻烦。现在，负载均衡器都可以配置成使用粘性session，也就是说，同一个用户的后续请求都会到同一个后端服务器。

在负载均衡器上启用粘性session只能解决一部分问题。现在，流量可以路由回拥有session数据的节点。但如果这个节点宕机怎么办？我们肯定不希望就此丢掉那部分session数据。为了解决这种问题，众多Servlet容器的生产厂商提出了不同的集群解决方案。而在集群间复制session会产生额外的挑战，稍后我们会谈到这些问题。

8.1.2 依赖容器

在很久之前，各种层出不穷的Servlet容器在Java Web应用开发中占据了很重要的位置。这些服务器对我们大有裨益，处理了一些底层的问题，比如`socket`、`main`方法等。如果代码用到HTTP Servlet API，就必须运行于某个容器中，理想情况下，我们应该避免调用任何厂商特定的API，保证Web应用是可移植的。虽然在很大程度上来说，只依赖Servlet API的应用可以在不同容器间保持可移植性，但是对于大多数人来说，每创建一个Java Web应用，就要用到一个容器。

对于运营部门来说，原本他们只知道如何支持其他种类的Web应用，现在他们还得搞懂Servlet容器。对他们来说，搞懂应用程序的行为已经是一种负担了，现在再加上一个Servlet容器，学会支持和监控应用程序，致使负担加重。

8.1.3　违反HTTP规范

很早之前，Java Web应用的开发者就发现，由Sun制定并由很多容器实现的API过于繁琐，使用也不方便。有鉴于此，一时间涌现出很多框架，尝试把这些API在最终用户面前隐藏起来，取而代之提供另一套API，在理想情况下，新的API开发者用起来会更容易。像Struts和WebWork这样的开源项目变得很流行，后来很多框架也试着提出构建Web应用的不同方法。但问题在于，因为这些框架的实现方式是隐藏的，所以很多程序员对Web——或者说HTTP的实际工作方式一无所知。通过把这些API隐藏在后面，由框架作者决定HTTP的哪些部分得到支持以及支持方式，虽然这一领域的许多开发者得到了帮助，却也造成了许多行为糟糕（以HTTP的角度）的应用。

以标准来看，行为糟糕的Java应用无法正确地设置cache头，而且会使用POST做一些奇怪的事情，很少发送500应答码表示错误。

很多现在流行的Java Web框架与更为现代的HTTP感知平台，比如Webmachine[①]（以它创建的应用绝不会不兼容HTTP规范）相比，差距是非常明显的。

每个Web应用程序员都应该理解HTTP规范，这对于开发出现代Web应用至关重要。有一些模式，比如POST重定向到GET、按新鲜程度分区等，可以帮助我们构建行为良好的应用，解决Web应用的一些常见问题。

8.2　无状态服务器

无状态服务器是一组技术。

8.2.1　集群

前面提到过，很多Servlet容器都提供集群功能，它可以把session数据复制到多个服务器节点上，以此将一个节点宕机的数据损失降到最低。

有了集群，用户session状态就可以在每个集群的两个或多个节点间复制。由于数据复制可能会有延迟，我们还是会用到粘性session负载均衡，但至少如果原来的节点宕机了，用户还应该重定向到一个持有其session数据的节点。虽然设置集群并不是特别复杂，但也绝非轻易之举——一般来说，开发人员不会在自己的开发机器上运行这样的环境。这也意味着，一些问题（比如尝试在节点间分布不可序列化的session）通常很晚才能发现。

配置成本固然高昂，但更令人担忧的是在集群所有节点分布整个session的开销。这会引发负

① https://bitbucket.org/justin/webmachine/wiki/Home

面的网络效应，复制大型session对象会占满内部网络连接，造成整个站点多次停止响应。这一网络效应使得添加节点进行扩展越来越难，而跨数据中心的集群根本就不现实，一个数据中心的故障转移通常会导致session数据丢失。你现在应该开始理解为什么Mike Nygard在*Release it!*一书中说“session是Web应用的阿喀琉斯之踵”了吧。

像Gemfire和Terracotta这样的分布式缓存系统供应商，一直标榜其session状态复制比Servlet容器集群的标准功能更有效。尽管分布式缓存有很多好的实施案例，其中也确实有些优秀的产品，但用它们尝试绕过传统session复制的问题还是没有命中要害。处理缓存失效已经很困难了，数据缓存的地方越多，理解如何处理缓存数据失效就越复杂。如果只是用分布式缓存解决session复制，也许应该仔细思考一番，是否从一开始就应该规避在服务器上存储数据的需求呢。

8.2.2 cookie救世

如果做过这样的Web应用，能够认证用户并创建session，那你一定创建过cookie。无论是手工创建，还是由Web框架创建，从实现上说，反正都创建了cookie，用以让应用服务器知道“这个人登录了”。以前，Servlet容器在创建`HTTPSession`时会生成一个含有ID的cookie。要取得后续请求的session状态时，就用cookie里的ID在Servlet容器的session存储里进行查找。

与只是使用cookie存储ID相比，我们还可以把与session状态相关的各种信息都放到cookie里。稍微想一下，就会知道我们可以取出所有关心的数据，把它们放到cookie里，而不是依赖服务端状态，而且还得要复制。

请求里有了session状态，后续请求需要到某个特定机器的需求也就灰飞烟灭了。如果就算上次访问的节点宕机了，而且下一个请求到了数据中心的另外一台机器上（甚至故障转移到另外一个数据中心），我们也根本注意不到。我们也无需在负载均衡器上开启粘性session，应用服务器集群的大部分需求也不复存在了。

很多大型网站都理所当然地用cookie做session状态。这里的关键在于，它们希望应用服务器能够具备以线性方式扩展的能力。与在服务端管理数据的复杂度相比，使用cookie做session状态的复杂度可以说是微不足道。如果能够做到没有应用服务器集群，故障转移会简单得多，扩展也更轻松，如果不再需要购买那些以“立即提升集群支持!”为口号的Servlet容器，还能帮我们省下一笔钱呢。

你是否对那些只用Apache服务器（还看不到集群容器）就能轻松支持百万用户的PHP应用感到好奇呢？那么现在你应该知道其中的部分奇妙之处了。

8.2.3 区分用户特定的数据

我们需要知道，cookie虽小，却可累加。如果我们在www.mycompany.com域下放一个cookie，

客户端会把这个cookie发给所有发到该域的请求中。其中会包括并不关心用户准确状态的请求（也许甚至不关心用户是否是认证用户），比如静态图片或CSS文件。如果有可能，请将流量分流，只把session状态的cookie放在那些需要session状态的子域下，比如我们不会给static.mycompany.com发送cookie，但会给myapp.mycompany.com发送。

8.2.4　安全cookie

如果要把session状态放到用户cookie的迁移中，需要明白怎样使用cookie。如果cookie被某个不怀好意的人窃取了该怎么办呢？这或许不像我们想象中的那种理论上的风险；比如每当我们通过不安全的无线网络（如机场休息处或是咖啡店）使用HTTP时，其他人可能会拿到我们的cookie。如果有人用我们的cookie发送请求，他们就可能窃取我们的session数据。实际上，Firefox有一款插件Firesheep[①]，它充分展示了这是一个多严重、多现实的问题；我们可以用它浏览本地网络中发送的所有cookie。

避免此问题的一种方式是，通过HTTPS发送所有数据，但HTTPS会增加流量负担，而且不只是花费客户端和服务器端的处理时间，还有带宽的负载。对于大型网站而言，这种负担是非常大的——甚至大到我们需要限制HTTPS的使用，只用它处理需要额外保护的数据。

因此，我们可以只通过HTTPS发送session状态，所有其他流量则通过HTTP发送。当验证用户身份时，验证期间的cookie用`Secure`[②]属性来写，这可以确保客户端只能通过HTTPS传输该cookie，这样cookie就不会被盗用了。

然而，某些针对用户的内容即便被劫持，也不是什么严重问题，而且这些内容是可以通过HTTP发送的，可是怎样做到这一点呢？和前面一样，访问真正安全的内容需要有`Secure`属性的cookie，而且只能通过HTTPS发送，但是用户特定的、安全要求不高的内容只要用标准的、非安全的cookie通过HTTP发送即可。我们要做的只是在认证过程中把这两种cookie都发过去就好了。

举个真实的例子，比如说一个购物网站。HTTP cookie已经有了，也许声明周期还很长（可能有几周）。这种cookie的存在让网站可以向用户做一些个性化推荐，几乎还能根据用户资料显示“Welcome, Joe Blogs!”之类的消息。但如果要查看订单，我们是无法容忍别人冒充的，所以需要查看是否存在安全cookie，把我们转到HTTPS上。这种安全cookie的生命周期要短一些，以免有人使用别人电脑访问到cookie。如果安全cookie不存在，我们就要重新登录了（当然是通过HTTPS登录）。这样的话，最糟糕的情形就是有人能够访问我们的购物车，但不能结账，也不能访问账单信息。

---

① http://codebutler.com/firesheep

② http://tools.ietf.org/html/rfc6265#section-4.1.2.5

8.3 容器是可选的

虽然容器能提供一些有用的特性，比如说集群、监控和控制，但这些特性却在开发代码时助益不多。容器的存在似乎反而会阻碍应用程序开发和测试。如果没有容器特定热部署技术，反馈周期真的会长到影响团队的效率，有的容器很难安装和自动化，因此想把其作为持续集成构建的一部分就很困难。另外，如果稳妥运行测试的唯一方式是启动整个服务器栈，那么构建的时间也会变得很长。

8.3.1 容器外测试

Python的WSGI API受到了Servlet API[1]的启发，但与Java不同的是，它以wsgi_intercept[2]的形式为API提供了一个可行的mock层，这样测试可以运行在一个mock的HTTP传输上。这就极大地减少了测试的时间。如果应用程序实现采用了Servlet容器内外都可用的API，Python Web程序员从wsgi_intercept中得到的那些益处，我们也可在Java中享受得到。

行文至此，我们可以稍做转移话题，谈谈测试。总体来说，我们需要在编写不同类型的测试之间寻找平衡。Mike Cohn为此提出了一个很好的模型，他把不同类型的测试映射为一个金字塔（见图8-1）。不同版本的Mike金字塔显示不同的层次，有的是单元/服务/UI，有的是单元测试/验收测试/GUI测试，等等。我见过的最好的版本是按照测试覆盖范围的大小划分测试的，故而在本例中，显示为大中小三个范围的测试。

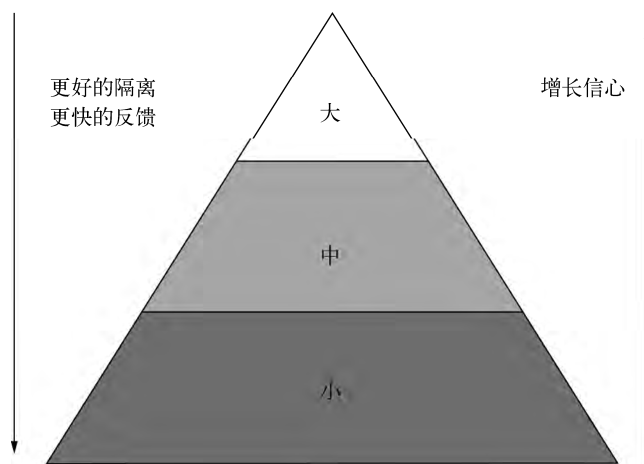


图8-1 Mike Cohn的测试金字塔

① http://www.python.org/dev/peps/pep-0333/
② http://code.google.com/p/wsgi-intercept/

不同项目常常赋予测试不同的名字，但其适用原则不变。金字塔最底端的测试覆盖范围最小。这种测试是我们做TDD（这是在ThoughtWorks必做的事情）时创建出来的。这种小范围测试只会覆盖很少的功能点——通常只是一个方法。这种测试运行很快，当测试失败时，我们通常可以很准确的知道到底是什么出问题了。不过，这种测试的本质决定了每个测试只会覆盖系统的很小一部分，所以即便一个测试通过了，我们也未必就有信心认为系统能按预期工作。

金字塔中间的测试覆盖范围要更大一些。在Web开发世界里，金字塔顶端的测试会端到端地执行系统行为，典型情况下，这里会用到一些驱动器技术，比如Selenium或Waitir。可能你已经想到了，这种测试通过时，我们对系统能按照预期工作就更有信心了。

大范围测试的缺点在于运行时间很长，造成这种情况的原因很多。通常，用真实浏览器驱动测试会导致速度变慢。比如，考虑下面这个典型的例子，它用到了源自Selenium 2的WebDriver API，下面用Firefox搜索“Modern Java Web Applications!”：

```
WebDriver driver = new FirefoxDriver();
driver.navigate().to("http://www.google.com");

WebElement element = driver.findElement(By.name("q"));
element.sendKeys("Modern Java Web Applications!");
element.submit();
```

上述例子启动了一个真实的浏览器用以驱动测试。但我们真的需要真实的浏览器吗？我们在测试复杂的、浏览器特定的行为吗？Selenium 2提供一个HTMLUnit抽象，可以在模拟浏览器上运行测试，用Rhino可以模拟浏览器的JavaScript执行。Rhino做得很不错，可以支持像JQuery这样复杂的JavaScript应用程序。那么要怎么做才能摆脱浏览器税呢？其实只要把`FirefoxDriver()`换成为`HtmlUnitDriver()`即可。总之，尽量避免过多浏览器内的测试，将使用模拟浏览器进行测试作为默认选项。

另一个导致浏览器测试缓慢的因素是，一开始就需要通过套接字进行连接。如果不用真实的浏览器，我们是否也能摆脱这个需求呢？不幸的是，目前所有的WebDriver实现都假设我们连接的是远程的URL。这就必须在某个地方通过套接字连接测试的端点（endpoint）。为了绕过这个限制，我在ThoughtWorks同事Alex Harin等人创建了Inproctester[①]。它让我们仍然可以用WebDriver的API写测试，但无需通过套接字通信，我们可以直接与内嵌的Servlet容器的底层API通信；目前，Inproctester支持Jetty和SimpleWeb。

这样做的好处是什么呢？如果我们决定用真实浏览器的运行容器内启动同一套测试，只要把容器启动起来，在测试里替换不同的WebDriver实现即可。

结果如何呢？在实践中，我们的测试速度得到了数量级的提升。能够节省多少时间，取决于

① 参见https://github.com/aharin/inproctester以及由Jennifer Smith和其他ThoughtWorker创建的相关的.NET程序库Plasma，这些程序库实际上要早于Inproctester（https://github.com/jennifersmith/plasma）。

测试所做的事情，在我参与过一个项目里，我们一般期望容器测试能在1秒内完成，结果我们做到了，在一台标准的台式机上，4分钟之内运行了1000多个测试。在开始新内容前，最后要说的一点是：这些都不是什么全新的东西。Python程序员多年之前就在用WSGI和Twill做同样的事情了，但时至今日，Java和.NET的Web开发世界里，也没有多少人这样做。

8.3.2　我们真的需要容器吗

我们已经确定可以用无状态的方式创建一个Web应用，这就完全规避了创建集群的需要。为了进行高速测试，我们可以在容器外运行所有的测试。这样一来，我们还需要容器吗？喔，在很多情况下，答案是不需要——不过有一些轻量级（但仍兼容HttpServlet API）容器也能以嵌入式模式运行，比如Jetty。我们只要运行`main`方法就可以为应用启动轻量级的Web服务器，而无需将应用程序打成WAR文件，部署到一个已运行的容器中。这样的应用程序无需商业许可，而且通常运行得像有容器绑定的一样快。

如果以嵌入式容器方式运行，就无需web.xml、WAR文件结构或是其它东西了。我们可以用代码自己绑定整个应用（如果我们愿意的话）。我们也可以用onejar[1]把整个应用服务器打包到一个可执行的JAR文件中去；然后只需要运行`java -jar myapp.war`，应用服务器便可启动。

不用容器运行运行Web应用通常可以简化部署流程，如果不是特别依赖Servlet容器提供的特性，我强烈建议你尝试一下不用容器的运行方式。

8.4　按新鲜程度分区

按新鲜程度分区是另一组技术。

8.4.1　缓存：可扩展网站的秘密武器

在ThoughtWorks的内部邮件列表里，最近有人问了一个问题："面对一个大规模基于Web的系统，你会选用哪款Servlet容器？"我差点就脱口而出："用不到的那个Servlet容器。"如果应用程序要做的工作很少，处理的请求很少，也就无需考虑寻找更快的容器、更强大的机器或是更多的机器。第一个想到的"用不到"的答案是什么呢？其实是缓存。

任何一个用户浏览的网站，我们至少已经在用浏览器缓存减轻网站负担了。

浏览器缓存（在理想情况下）会根据应答头里的信息决定更新已有的内容的频率。一旦浏览器决定再次检查内容，它会发送一个有条件的GET请求；如果网站能够正确返回304 Not Modified

① http://one-jar.sourceforge.net/

应答（请记住：了解HTTP规范。），浏览器会继续使用本地缓存的内容。

在现实中，浏览器和服务器之间可能还有别的缓存系统。我们可能会用到CloudFront或是Akamai这样的内容发布网络（Content Delivery Networks，CDN）。ISP和企业网络可能也会缓存。你可能也会用Squid、Varnish或是Nginx做反向代理。设置正确缓存头的好处在于，我们几乎可以免费使用这些系统。

8.4.2　选择缓存的内容

我们从一个简单的例子开始。假如在一个新建网站上有一篇文章，网站的大部分内容都是静态的——网站装饰（比如CSS、JavaScript和图片）只在两次发布之间才可能变化，文章本身也只会有几次微调的机会。不过，作为现代化的新网站，它通常会包含很多更动态的、更面向用户的内容：比如最新发布的5篇文章的列表，突发的新闻简讯，给登录用户提供报价信息。

我们想要用缓存减少用户从服务器上请求该页面的频率。网站装饰可以由单独的请求加载，所以它们可以用时间较长的缓存头。不过HTML本身怎么办？它混合了相当静态的内容（文章）和需要刷新的内容。我们可以设置一个较短的缓存头，以此确保动态内容的新鲜度，但这样一来，我们还会取回大量未变的内容。

那我们该如何满足静态内容有较长的生存期的同时，还能保证动态内容得到更新呢？按新鲜程度分区[①]，这正是我们需要的答案。

8.4.3　按新鲜程度分区简介

按新鲜程度分区将页面分成若干片段，最后再组成完整的页面。片段是根据内容新鲜度进行分区的。在我们的新闻示例里，可能有两个片段：文章和“浏览量最多的文章”小部件。

这些分区可以在服务端或客户端聚合成最终页面。如果是服务端聚合（见图8-2），一个请求会取得多个片段——有的来自缓存，有的来自服务器本身，然后组成最终的页面，并通过网络返回到用户的浏览器。我们可以重用基础设施里既有的反向代理扮演页面片段缓存的角色。

服务端内容聚合的缺点在于，理想情况下，我们想让客户端从本地浏览器缓存取得文章体，而只从服务器获取最新、最流行的小部件内容，但这种做法却要把二者的流量都导到服务器上。因此，客户端聚合（见图8-3）可以解决这个问题。采用客户端聚合，发给客户端的HTML页面会有一系列JavaScript负载，它们负责从浏览器发起请求来回应一些额外的内容。浏览器则只是发起GET请求，得到所需的不同片段；只要那些GET应答的缓存头设置得当，浏览器就可以用本地缓存的版本。

---

① 在这一模式广泛地运用于ThoughtWorks的两个大规模项目之后，Martin Fowler最初已经有所记录，但和所有模式一样，它肯定在此之前就已经得到了广泛应用：http://martinfowler.com/bliki/SegmentationByFreshness.html。

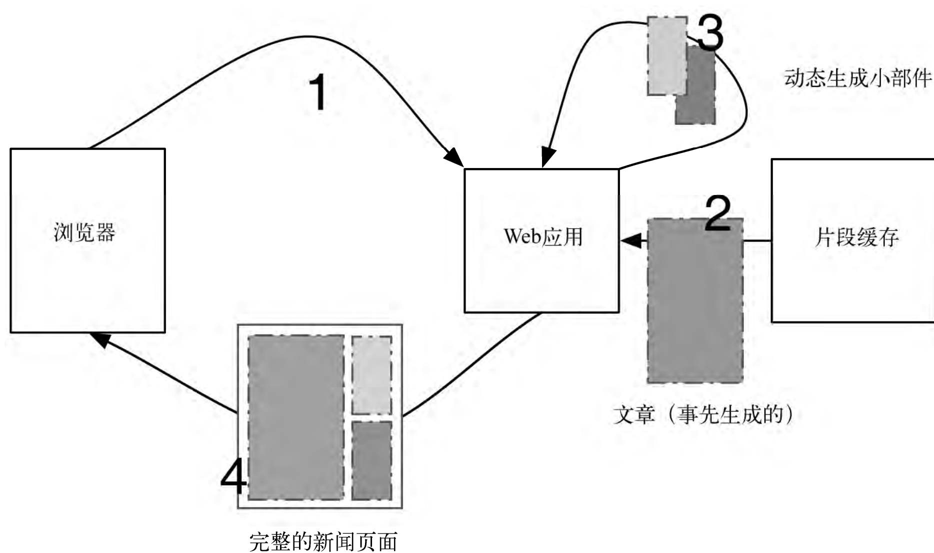


图8-2　服务端内容聚合

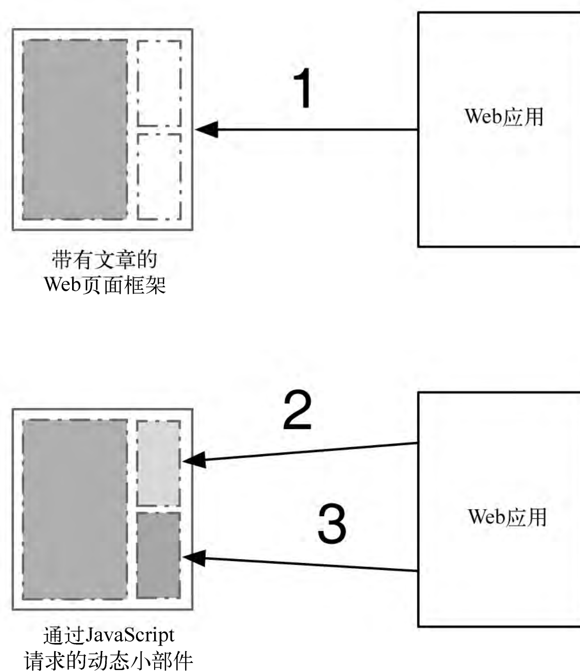


图8-3　客户端内容聚合

客户端内容聚合的缺点在于必须使用JavaScript（或者类似的客户端技术）。如果页面内容大部分由客户端技术组成，客户端的负担过重，可能会引起性能问题。很老的机器/浏览器执行大量JavaScript时会很吃力，即使服务器端很“乐意”加载，也会导致页面看上去加载得很慢（即页面上很多部分是空白的）。另一个缺点是，由JavaScript构建的页面部分是搜索引擎不可见的；不过，如果我们能保证搜索引擎索引的内容能在最初的静态HTML负载里，还是可以得到所需的搜索流量的。

两种聚合方式各擅胜场，因而我们也许要混合使用两种技术。可能会影响正确做法的因素包括：用户的位置、所用的浏览器以及计算机的类型，所以我们要反复调整，找到适合的方法。

渐进增强

使用JavaScript组装页面（客户端聚合）对于实现渐进式增强很有助益。如果网站要支持多种客户端，就需要考虑到不同浏览器具备的能力。有些网站会实现两套（或者更多）页面：一套给全功能浏览器，另一套给更初级的浏览器，比如IE6和屏幕阅读器。Steven Champeona* 就是渐进增强（也称为优雅退化）的提出者，他用这个词形容这样的网站：先加载基本HTML页面，然后执行JavaScript，再用更高级的UI特性、更丰富的内容等对HTML进行修饰。较初级的浏览器不会执行JavaScript增强页面，因此不会看见不能执行的代码。关键在于，基本的HTML页面能够让用户使用网站的核心功能；就比如购物网站，用户还是可以在上面结账的。

* 要了解更多背景，请参考http://www.hesketh.com/thought-leadership/our-publications/progressive-enhancement-and-future-web-design。

8.4.4　反向代理和内容发布网络简介

像Varnish和Squid这样的反向代理，以及像CloudFront和Akamai这样的内容发布网络（content delivery networks，CDN），都部署在网站之前。在最简单的配置下，它们会遵守内容的缓存头信息，会以快速且优化的方式把HTTP应答缓存到内存中。所有的请求都要通过这种类型的缓存，如果内容在缓存不存在中，或是缓存已经过期，该请求才会导向到之后的Web应用去。

CDN和反向缓存主要有两点不同。第一，CDN一般由第三方提供，部署地点遍布全球，但反向代理一般和Web应用部署在同一个数据中心。第二，CDN通过DNS查询确保请求导向与其发起处最近的缓存节点。比如说，我要访问的网站服务器位于美国，但CDN在英国前置了一个离我较近的节点，这样缓存的内容就是从我的国家得到的，速度更快、延迟更低。

在反向代理和CDN之间做选择时，主要的考虑因素是价格和用户分布。Varnish、Squid和Nginx都是免费的（不过要自己准备硬件），而CDN通常是商业工具，收费额度通常由占用的带宽决定。如果用户群分布在距离数据中心不远的地方，使用CDN则得不偿失。另一方面，如果用户群遍布全球，可能会要用到托管CDN的高级解决方案。

反向代理和CDN都可以进行不同程度的配置。因而有些人不禁在缓存本身配置缓存规则，而不是最开始在应用程序里设定正确的头信息。我强烈建议，不要在这些缓存里配置内容的缓存时长。首先，网络里还存在其他类型的缓存，它们会从应用程序里设置的头信息中获益。其次，通过在应用程序里保存缓存行为，我们在切换不同的反向代理或CDN时，就能将调整幅度降到最低。

8.5 POST 重定向到 GET

任何需要处理POST请求的Web应用都会遇到一个常见的问题——处理重复请求。为了了解这个问题，我们讲一个简单的例子。

实例：购物车

用户在浏览购物网站时，会不时把货物加入购物车。最后，用户点击购物车上的“购买”按钮。这时会向服务器发送一个POST请求：购物车中的内容存在POST的参数里。一旦服务端处理了订单，就会渲染一个收据放在POST应答里返回给用户。

用户把返回的收据收藏起来。稍后，用户通过收藏夹重新访问该页面，结果却发现收据不见了。这是为什么呢？

这个场景的问题在于，收藏页面时，我们收藏的只是URI，而不是POST的参数。当然，实际上我们也不想收藏参数；否则，重新访问收藏页面时，我们可能（取决于应用程序的设计）最后会重复提交订单。我们真正想要的不过是看一下收据而已。

同样，我们多久会遇到一次这种现象：只是点击刷新，却被问是否要重新提交表单参数？如果这么做了，我们的预期又是什么呢？对用户而言，发送两次同样的请求有意义吗？

POST重定向到GET模式[1]就是用以规避这种问题的。一旦服务器处理完POST请求，它不是直接渲染应答返回（比如，上面例子中的收据），而是给浏览器发送一个重定向，告诉它到哪里去获取结果。稍后，浏览器发起一个GET请求到最终的重定向地址（见图8-4）。

浏览器重定向到的资源既可以刷新，也可以收藏，不必担心出现前面列出的问题。

如果网站的工作流很长，这种做法还有一个额外的好处。每一步都可以通过GET返回一个可操作的状态，而且可以收藏，这样工作流就可以在某个中间点继续（前提是应用程序允许这么做）。

① 参见Wendy Chisholm和Matt May合著的*Universal Design for Web Applications* [CM08]一书的36页，以了解更多细节。



图8-4　Post重定向到GET

HTTP应答码简介

HTTP 1.1*规范规定，POST重定向到GET的例子里，要发送的正确的重定向应该是303 See Other。303会告诉浏览器（或任何其他客户端），不同URI下的应答，应该执行GET请求到这个给定的URI获取应答。在这种情况下，许多应用程序使用的是302，HTTP 1.0规范将它归类为Moved Temporarily，大多数浏览器解释它的方式等同于303。303是在HTTP 1.1规范里引入的，消除了不同实现处理302的差异，在此不再赘述。

* 参见http://tools.ietf.org/html/rfc2616#page-63。

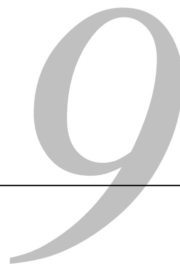
8.6　总结

如本章所述，除了标准方式之外，构建Java Web应用程序有诸多不同的技术。以我们的经验来看，无论是最终系统的可用性，还是开发人员的工作效率，这些技术都能带来极大的提升。此外还可以降低软件成本，减少运维开销。

本章所列的许多技术既可以独立使用，也可以组合起来使用。重要的是我们要意识到，标准和厂商做事的方式俨然不再是最佳的了。到本书付梓之际，世界各地的天才们很可能又在这个领域做出了各种有趣的改进——发现它们、付诸尝试、找出最适合自己的方式，一切都取决于你自己。

第 9 章

驾驭集成难题



Julio Maia撰文

做增量式交互开发时，每个集成点都是挑战。当系统中存在许多集成点时，我们必将面临一系列的挑战。

理想情况下，我们希望系统的每一处修改都能在集成环境中进行测试。但实际上，这一点很难做到，因为集成问题可能是不稳定的、数据变化很快、速度很慢，甚至根本不存在集成环境。

因此，我们被迫在一个独立的环境中进行测试，以期得到快速反馈。虽然这并不足以验证系统的功能，但只要前面没有出错，在集成环境中就可以快速发现系统中的问题，并将问题本身从真实的集成环境中分离出来。

我们需要分离关注点，定义好集成契约。这就要求小心仔细地进行模块化，部署全面的测试策略。

虽然标准的模块化和组件化技术可以用于分离关注点，但仅这些还不够。重要的是，通过测试定义可执行的集成契约，确保子系统的实现细节不会影响主要的集成系统。

而测试就可以看作是各利益方和团队之间，围绕系统及其集成点的预期行为所达成协议的协议载体。

我们想要创建和维护测试基础设施，并通过模块化的方式实现，支持分散而高效的交付。其中难免会遇到障碍。因为无技术背景的利益方会认为，这些工作和特性没有直接关系，然后不断降低这些工作的优先级。

与其和他们正面冲突，不如将由快速反馈带来的生产力提升显示出来，让所有的利益方都能感受到开发和维护这些框架的重要性。

我们通过收集重要指标表现出生产力的提升，所以也很有必要将收集指标的工作也看做开发工作的一部分。

9.1　持续集成方法

为了验证系统的功能，有几类测试需要在特定的集成环境下执行。比如集成测试，功能/验收测试以及性能测试。在实际情况中，这些测试环境往往具有以下特点。

- **不稳定：**环境的正常运行时间得不到保证，或者因性能或行为变化而出现随机性的超时和前后不一致的响应。
- **运行缓慢：**响应时间缓慢，无法满足自动化测试的需要；构建可扩展性需要大于集成服务器所能提供的处理能力。
- **并非总是可用：**由于提供服务所需的资源成本过高，或者只在特定的时间段可用，这些测试环境可能无法一直处于运行状态。
- **数据变化频繁：**由于种种原因，测试数据可能会不断变化，但要想避免这种情况发生，成本又会令人望而却步。
- **集成环境根本就不存在：**各个团队要一起努力建立一个将来某个阶段才会集成在一起的方案。

在这种情况下，持续集成必将举步维艰。因为测试会频繁失败，而且构建成功与否将不单单取决于系统本身的修改，还可能是集成环境本身的问题，这样就很难找到问题的原因了。

这些环境问题可能给测试带来很大影响，但总是有办法应对的。要做到这一点，我们需要有个稳定基准，它要符合如下要求：

- 一组快速且可扩展的本地构建；
- 创建测试在集成环境中所需的基准数据；
- 验证系统可以在基准数据之上正常工作；
- 快速精准地定位导致测试在集成环境中失败的原因。

这个稳定的基准环境需要在构建流水线中增加一个额外阶段，而且该阶段是完全独立的，因为它不依赖任何外部环境。

9.1.1　稳定基准

采用以下实现方式可以不依赖集成环境进行集成测试和功能测试。

- 使用测试替身，它会实现与系统中真实组件一样的编程接口，但无需调用外部集成点。比如，无需和数据库打交道，就可实现DAO提供静态数据。
- 使用服务器，使其提供与集成环境中真实服务器相同的数据、行为和协议。这些服务器不见得是测试用的模拟服务器，它们也可以是适用于生产环境的服务器（例如，本地部署使用与集成环境相同的服务器）。

那么，以上方法有何不同呢？

测试替身要求创建包含有测试代码的产品包。这并非我们的初衷，因为这样增加了构建过程的复杂性，并且构建流水线违背了单一产品包原则。此外，由于没有和外部系统交互，测试替身对集成测试和功能测试的价值也不大。

因此，更可取的稳定基准方案是，提供与集成环境相同行为和协议的服务器。这种方法可能需要定制软件，模拟集成服务器的行为，但其优势是能够运用到整个集成栈，也不需要在产品包中添加测试代码。（从总体上看，对于构建流水线和测试而言，这是一种好的实践。）

9.1.2　集成stub

为了验证系能够正确集成，却又不想完全复制集成点，一种有效的方式是使用集成stub模拟每个集成点。这一方法需要创建与真实服务器行为完全相同的服务器。（虽然只是用做集成测试的夹具而已。）

与集成环境相似，集成stub提供相同的数据、协议及语义。仅由stub组成的独立环境不依赖外部服务器。无论stub是否运行在相同的应用程序空间中，都必须与应用程序代码分离开来。

通过下述几种策略，可以实现stub集成。

- 在本地部署中，使用与集成环境相同的软件，或者这些软件的简装版（如使用Oracle express来替代Oracle）。
- 使用协议兼容的服务器（比如，内存数据库、SMTP服务器和FTP服务器等）。
- 使用原生服务器，可以采用不同的策略获取数据。
  - 录制重放：当测试在集成环境中运行时，使用录制代理或者应用程序钩子（hook）创建数据流快照（如使用录制/重放代理）。
  - 固定数据：使用手工创建的测试数据集，可通过查询集成服务器或其他方式得到这些数据。
  - 重新实现规则：实现与集成服务器完全相同的语义规则。一个具体的案例就是生成器，它采用已知顺序提供数据。

那上述实现策略有何不同呢？

本地部署产品兼容的服务器有时的确很方便，但需要开发人员在自己的的机器上安装并配置服务器（或者依赖一个特定的开发机，这会带来单点失败问题），以便对系统做出修改。这未必是一个大问题，而且也得取决于软件安装的复杂程度，以及这些服务器为了完成测试所需的资源数量。另外，这种本地部署有一个显而易见的优势：能够提供良好的兼容性。

9

协议兼容的服务器通常运行在内存中，虽然运行良好，但也可能存在兼容问题，即便在早期开发阶段也难以预见。这些服务器通常都用来作内存服务器，不需要本地安装。

而原生服务器的构建或设置就会有些难度。通常说来，这对只读系统来说并不复杂。然而，事务性系统要求状态管理，进而要求一个详尽的策略区分和记录事务数据。但另一方面，它们完全由开发人员控制，构建通常是轻量级的。

无论选择哪种实现，要创建独立的stub系统，关键是所有stub都需要自身的集成测试。这一点无法避免，因为这样能保证这些系统按预期运行，也排除了stub系统实现中的重大缺陷导致主应用程序代码失败的可能。

9.1.3　构建流水线

如果可以保证持续地在集成环境中测试，就可以只用独立的环境运行本地构建。应当尝试在该环境中运行所有的集成测试，而不是在本地只运行全部测试的子集，依赖持续集成服务器运行所有测试会导致构建频繁失败。对于有大量测试的系统，在本地构建中运行所有测试，而不使用并行执行是不现实的。使用轻量级的内存stub，让测试分布在不同的机器上运行，使构建更具可扩展性。

如果正确地实现了模拟环境，基于这个环境的构建永远不失败。这意味着，流水线上的构建不会再与一个失败的独立环境关联在一起。

如果做到了这一点，基于真实集成环境的测试就可以安全地推迟到流水线的后几个阶段执行。这时如果有测试失败了，则有以下几种可能：

- stub未能提供与集成服务器一样的协议和语义；
- 集成服务器中的数据发生了变化；
- 集成服务器不可用或行为不一致。

多数构建失败的原因可能都是上述的第二、第三种情况。以集成方式运行时，稳定基准能够准确定位应用程序代码中的大多数问题，但却无法识别集成环境中的问题。这意味着仍然需要某种方式隔离真实集成环境中的问题。

9.1.4　监控器

构建流水线所使用的集成点检测器是一些专门的进程，它们会不断地收集和展示集成点的状态。在集成阶段，理解构建流水线出现的问题是很重要的，这些检测器可以帮我们判断问题是由软件修改所导致，还是因为stub的问题，抑或是集成点的问题。

要区分构建流水线可能发生的问题，有两种监控器很重要。

- **验证监控器**（verify monitor）：为了检查stub提供的数据与集成服务器提供的数据是否匹配，应当对每一个stub都实现验证功能。该功能可以采用stub录制或存储的数据，以此向真正的集成点发送消息，检查收到的应答数据是否与stub提供的应答数据一致。
- **可用性（心跳）监控器**（availability（或heartbeat）monitor）：我们可以设置一组监控器，检查集成环境是否可用，是否有响应。可用性监控器会向每个集成点发送一些简单的查询请求并度量响应时间（包括根本没有响应的情况）。这些监控器通常和用于监控产品环境的监控器一样。

这些监控器也可以用于度量停机时间，以及交付中吞吐数据变化所带来的影响。采用监控器，我们可以度量处理外部系统问题所需的成本，并将这些鲜活的数据提供给业务决策者，从而提高集成点问题的优先级。

9.2 定义集成契约

正如上节所述，处理集成点需要仔细地构建一套基础设施，提升从开发到支持整个生命周期的可见性和循环周期。与此同时，管理不同团队的交互也很有挑战。

分离不同系统之间的关注点有助于创建一个可管理的环境。共享组件也有助于将集成的复杂性最小化。然而，想要维护一个可持续验证，并为所有利益方（技术或非技术的）所理解的契约，这些技术也难有用武之地。

有一种方式可以定义不同系统之间的预期接口，提高系统在集成方式下当前状态的可见性，那就是在系统间建立可执行的契约。

不难发现，一些验证集成组件实现的集成测试，仅仅是写给开发团队使用。尽管这种方式的确有一定价值，能够持续验证集成点之间的某些期望结果是否满足，但对于外部团队却没有任何意义，无法验证系统间的假设是否有效，无法向非技术利益方展示当前的交付状态。

可执行契约可以这样定义：将集成测试实现为带有描述的、可由不同团队理解的验收性测试。因为这些测试可以运行在持续集成环境中，它的结果报告就可以用来监测每个集成点的状态。

9.3 度量和可见性

很多集成相关的问题会减慢项目进度，提高项目成本。但没有直接方法跟踪这些问题，而只有理解了需要解决的问题，才能将项目的产出最大化。

为了区分外部系统集成问题方案的优先级，我们需要获取下述指标：

- 在集成环境中运行时，失败构建的数量；
- 哪些系统特性会经常由于集成问题而被破坏；
- 每个集成点的可用性和响应时间；
- 集成环境中测试数据的变化频率；
- 处理不同类型集成点问题的成本。

因为上述信息并非一次就能人工收集完成，所以持续收集这些指标的信息很重要。要产生这些指标信息，需要添加一些自动化功能，但通常来说所需的工作量很小，创建一个dashboard页面关联数据，以可视化的方式显示处理项目集成问题的影响，为此做一些投入是值得的。

9.4　总结

不幸的是，大部分项目与其他系统集成时并不顺利，许多修改都变得危险重重，代价高昂。因此，集成问题也影响到了以短反馈周期进行增量开发的能力。

在项目开始阶段着手处理集成点的成本通常会小很多，而且只要花费很少的注意力，就可以搭建测试和监控的基础架构，支持开发工作。但随着系统中集成点不断增加，与外部系统集成的相关问题也会急剧增加，这对能否在系统中安全地添加新功能会有严重影响。

一方面，非技术利益方不认为修复集成相关问题是高优先级的，因为他们不知道集成问题的影响面有多大。另一方面，技术利益方虽然了解问题的严重性，但他们往往面临交付压力，尽管集成问题才是导致开发越来越慢的根源，但他们也无暇深究。

交付一个与复杂外部系统集成的系统着实不易，但将问题最小化，创建一个可持续、可管理的开发环境却并非高不可攀，只是需要为之持续投入。我们需要不断地改进构建测试的基础设施，这样对系统的修改就可以得到快速的反馈，同时，开发过程中可能由于外部系统引发的具体问题也会显示出来。与此同时，在系统构建时，自动验证不同团队间的契约也是很关键的。最终，我们关注于创建一个基础设施，对于软件交付中什么是真正需要持续改进的东西，让开发人员和业务人员有着共同的理解和重点，而不再总是强调技术债。

第 10 章

实践中的特性开关

JCosmin Stejerean撰文

特性分支模式[①]是在多个特性并行开发时的常用方式之一，它使用版本管理工具的分支功能保证特性的独立开发，并在每个特性完成之后合并回主分支。

这种方式的问题在于，独立开发的代码分支不能长时间集成，因此损失了持续集成[②]带来的益处。分支从主干分出的时间越长，延迟集成将大大提升累积的风险，在尝试把分支合并回主干时，这个问题则更为明显。想象一下，合并几个月的工作听起来会有趣吗？

处理长时间分支的复杂合并，有一种略为简单解决方案，即定期将主干上的修改合并到分支上。不过，这种做法也就只能做到这一步。对于特性分支上所做的工作，在主干上工作的开发人员知之甚少，有些甚至一无所知。主干上的重构不得不手工合并到特性分支上，因为这些重构通常无暇顾及特性分支上的工作。

在特性分支上工作的开发人员做的重构只会使合并更复杂，最终合并工作单元的时间比原本花在开发上的时间还长。这样一来，开发人员难免会排斥引入可能引起复杂合并冲突的修改。因此导致必要的重构延迟，甚至彻底取消。技术债务随即大大增加。

另一种近来赢得不少关注的做法是，在主干上开发所有的特性。这种做法也叫做基于主干开发[③]。如果在开发特性的过程中，所有工作都可以在一个发布周期内完成，这种做法就会非常有用。如果一个发布周期内不能完成所有特性，未完成特性就必须关闭，以免暴露给用户。这就是特性开关模式[④]。

① http://martinfowler.com/bliki/FeatureBranch.html
② http://www.martinfowler.com/articles/continuousIntegration.html
③ http://jawspeak.com/tag/trunk-based-development/
④ http://martinfowler.com/bliki/FeatureToggle.html

10.1　简单特性开关

在模板上使用简单条件，显示或隐藏界面特性的入口点，就可以实现最基本的特性开关：

```
<c:if test="${featureFoo}">
    <a href="/foo">Foo</a>
</c:if>
```

简单修改一下应用程序的逻辑，也可以实现类似的功能：

```
public void doSomething() {
    if (featureFoo) {
        «foo specific logic»
    }
    «regular logic»
}
```

对于更复杂的修改，这种做法会产生连锁反应式的条件判断，影响到整个代码库。更有甚者，这些条件判断在发布之后会依然长存于代码库里，或是为以防万一而保留，整个应用最终会为嵌套的条件所淹没，导致代码无法维护，难以理解。

10.2　可维护的特性开关

对于大范围的修改，我们应该用继承或者组合来扩展代码，实现特性相关的功能，重构必要的地方，提供整洁的扩展点。

我们可以利用继承增加一个扩展点：

```
public interface Processor {
    void process(Bar bar);
}
public class CoreProcessor implements Processor {
    public void process(Bar bar) {
        doSomething(bar);
        handleFoo(bar);
        doSomethingElse(bar);
    }

    protected void handleFoo(Bar bar) {
    }
}

public class FooProcessor extends CoreProcessor {
    protected void handleFoo(Bar bar) {
        doSomethingFooSpecific(bar);
    }
}
```

或者使用组合达到同样的目的：

```
public interface FeatureHandler {
    void handle(Bar bar);
}

public class Processor {
    FeatureHandler handler;

    public Processor(FeatureHandler handler) {
        this.handler = handler;
    }

    public void process(Bar bar) {
        doSomething();
        handler.handle(bar);
        doSomethingElse();
    }
}

public class CoreHandler implements Handler {
    public void handle(Bar bar) {
    }
}

public class FooHandler implements Handler {
    public void handle(Bar bar) {
        doSomethingCompletelyDifferent(bar);
    }
}
```

10.2.1 依赖注入

大部分特性相关的配置都可以用依赖注入[①]容器实现。现在我们看看在Spring MVC中是如何实现做到这一点的。

我们可以为特性相关的bean定义添加单独的`applicationContext${feature}xml`文件。不过，在某些情况下，我们还是要处理bean的列表，例如拦截器的列表。在特性相关的环境（context）文件中重复这个列表将给维护带来麻烦。

因此最好还是把这个列表放在基础的`applicationContext.xml`文件里。当某个特性需要增加拦截器时，可以将其添加到主列表里，在特性相关的环境文件里定义bean，在核心环境文件里添加一个空实现[②]。

① http://martinfowler.com/articles/injection.html

② http://en.wikipedia.org/wiki/Null_Object_pattern

10.2.2　注解

我们还可以利用注解，创建一个特性标记注解，消除XML中的重复。注解的值用以表示特性开或关的时候，是否需要这个注解的组件。

FeatureTogglesInPractice/annotation/Foo.java

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Foo {
    boolean value() default true;
}
```

我们还要创建一个自定义的`TypeFilter`，它将使用特性开关的信息和注解包含正确的实现：

FeatureTogglesInPractice/annotation/FeatureIncludeFilter.java

```
public class FeatureIncludeFilter implements TypeFilter {

    private final TypeFilter fooFilter = new AnnotationTypeFilter(Foo.class, true);

    public boolean match(MetadataReader metadataReader,
                         MetadataReaderFactory metadataReaderFactory)
        throws IOException {

        if (fooFilter.match(metadataReader, metadataReaderFactory)) {
            boolean value = getAnnotationValue(metadataReader, Foo.class);

            if (FeatureToggles.isFooEnabled()) {
                return value;
            } else {
                return !value;
            }
        }
        return false;
    }

    private boolean getAnnotationValue(MetadataReader metadataReader,
                                       Class annotationClass) {
        return (Boolean) metadataReader.
            getAnnotationMetadata().
            getAnnotationAttributes(annotationClass.getName()).
            get("value");
    }
}
```

并将类型过滤器添加到Spring的组件扫描配置里：

```
<context:component-scan base-package="com.example.features">
  <context:include-filter type="custom"
```

```
                          expression="com.example.features.FeatureIncludeFilter" />
</context:component-scan>
```

现在，我们可以继续注解相应的实现，Spring将会处理剩下的事情：

```
public interface Processor {
   «»
}

@Foo(false)
public class CoreProcessor implements Processor {
   «»
}

@Foo
public class FooProcessor extends CoreProcessor {
    «»
}
```

添加更多的特性后会更加有趣。有时也需要扩展该类型过滤器处理复杂的特性组合。

10.3　分离静态资源

前面的章节展示了一些在服务端处理特性开关的方法，但如何处理像JavaScript和CSS这样的静态资源呢？

我们可以把一些静态资源转换成服务器端渲染的模板。通过这种方法，就可以添加逻辑，按照特性相关的方式修改这些静态资源。这种做法会把由CDN[1]托管的静态资源移回到应用服务器，在某些场景下，这是不可接受的。

我们还可以把静态资源转换成构建时渲染的模版。但这种做法可能也有问题，因为它限制我们必须通过重新部署才能开关特性。

不过有一种更好的做法，即将静态资源看作静态文件，对静态内容创建特性相关的版本，在动态模板中进行有条件的包含。因此，我们可以对shopping_cart.css创建一个shopping_cart_foo.css文件，Foo特性启用时，以此提供特定的样式。

就JavaScript而言，像服务器端代码一样，我们也可以在JavaScript函数内使用特性条件。不过，这种做法也有明显缺陷，它会把仍在开发、尚未准备发布的特性信息暴露给终端用户。有时意外泄露并非什么大问题，但在某些情况下，过早泄露没有发布的特性将会是灾难性的。现在来看看如何防止意外泄露未发布的特性。

① http://en.wikipedia.org/wiki/Content_delivery_network

10.4　阻止意外泄露

谈到保密性，在分离的分支上工作具有其天然的优势。这种方式完全可以保证分支上的代码在合并回主干之前都是保密的。过度担心意外泄露未发布的特性，是用特性开关替代特性分支的巨大障碍之一。

没有适当地包装特性开关，很可能会泄露未发布特性。如果幸运，它会以某种明显的方式自行报错。但通常差异很微妙：有可能是一些包含了错误文字的信息，或是某个额外的字段显示了出来，等等。

如果使用大量的手动测试，最终也会注意到不一致的情况。我们还可以尝试使用自动化功能测试，但是如果测试功能或者UI元素不存在会有些奇怪。

未发布特性可能泄露，其真正微妙之处在于不可见的东西，如未使用的CSS类，JavaScript代码，甚至是HTML注释。这种类型的意外泄露很难注意到。

防止意外泄露的最佳方式是遵循严谨的开发流程。任何作为特性的一部分而完成的工作，都应该由特性开关包装起来。修改静态资源应该在单独的特性相关的静态文件中完成，而且使用这些新文件也应当包在特性开关中。我们总是可以把特性相关的静态文件分开，如需要注意安全问题，可以不在部署中包含那些文件。

10.5　运行时开关

如果能在运行时开关特性，就可以部署一次应用，并且无论手动测试或是自动化功能测试集，都可以测试不同的特性。这样，如果在产品环境中出现错误，就可以迅速关掉特性。

关于运行时开关特性，首先要想一想如果用户正在使用，他是否能够立即见到特性关闭，或者特性开关的设置是否会在会话期间保持不变。

如果保持会话期间特性开关不变，那么用户在网站上就能看到一致性的体验。立即开关特性有个缺点，用户可能会对网站行为潜在的变化感到困惑。由于期望遭到破坏，还有可能会导致应用错误。尽管如此，立即开关特性在某些东西发生错误时，也允许我们在全局快速关闭特性。

灵活的特性开关系统既可以立即应用开关，也可以等到会话结束时应用，这取决于开关特性的紧迫性。

此外，我们还要考虑如何在多个应用服务器间传播特性开关。我们可以把特性开关放到集中的外部系统中，比如数据库或是文件中，并对其进行定期扫描。

我们还可以暴露一个管理接口，例如JMX[1]，继而可以针对运行中的应用服务器直接修改开关。这其中的好处在于能够立即更改，但也确实需要额外的协调，以确保修改能在整个服务器群中保持一致。如果要直接对运行中的应用服务器开关特性，还需要在应用重启时持久化特性开关。

构建时开关

在构建时开关特性也是可能的。构建时开关除了可以提供代码编译发布的优势，还可以用于特性修改依赖的版本，造成某些东西API不兼容的情况。

10.6　不兼容依赖

如果把依赖升级到新版本，但新版本完全不向后兼容，那会非常痛苦，痛苦程度则取决于修改范围。使用特性开关，这个问题可能会放大，因为我们会发现需要同时使用依赖的不同版本。就以同一个类有两个版本为例，它们可能有不同的构造器，不同的方法签名，或者完全不同的方法，该怎么办？

对这种情况，我们可以退回到使用反射，这样就可以动态调用正确的版本，通过欺骗静态类型检测器，还能编译通过。尽管如此，这样也会较为复杂，而且性能也会受到影响。

相反，我们可以为有差异的类创建一个包装器。我们可以从创建一个公共接口开始，暴露两个特性所需的所有东西，然后创建一个特性相关的实现，委托给适当的依赖版本。然后，把特性相关的实现隔离到单独的代码模块中，在构建时编译其中一个。这就是构建时开关，不仅有用而且有必要。

10.7　特性开关的测试

关于特性开关，一个常见的顾虑是，需要测试的内容有组合爆炸的可能。就理论上来说，组合数会随着特性数量成指数增长。但在实际中，这是很少出现的。只有对预期发布的组合进行测试才可以确定是否会出现这种情况。

实际需要测试的组合数量与我们使用的特性分支是一样的。只是前者更容易测试，因为我们可以在一个代码库完成测试。如果使用运行时开关，甚至还可以一次构建，根据不同测试的需要修改特性。

如果使用构建时开关，我们可以为每种组合单独创建构建流水线，对每条流水线采用不同的冒烟和回归测试集。每次提交都会触发所有流水线的构建。从CI的角度来看，最终的结果与单独

10

① http://en.wikipedia.org/wiki/Java_Management_Extensions

构建各个分支是一样的。

不过，独立分支与这种设置还是不同的，对特性A的提交可能会破坏特性B的测试或构建。公平地讲，独立分支还是颇有顾虑的。只不过在独立分支的情况下，这个问题推迟到了集成阶段。特性开关则在每次提交时都暴露这个问题，我认为这是有好处的。

10.8　删除完成特性的开关

采纳特性开关还会导致随着时间推移，代码里充斥着过时的开关。因此，删除不再需要的开关是很重要的。在特性完成并且部署到产品环境后，最好仍然保留开关几天或几周，直至确信特性行为是正确的。不过只要特性行为无误，就可以删除这个特性的开关，让代码关注应用程序的核心版本。

为了更容易地删除过时开关，我们可以把开关做成静态类型的。也就是说，创建一个`Feature-Toggles.isFooEnabled()`方法，而不是`Feature-Toggles.isFeatureEnabled("foo")`。这样，我们就能利用编译器，轻松地从代码里删除过时的特性，因为只要删除了`isFooEnabled()`方法，任何用到它的代码都会编译失败。

如果IDE支持，也可以让IDE找出给定方法的用武之地，继而就可以找到需要从模板删除的地方。

如果开发中的特性要无限期/长时间暂停，那么就很有可能使用开关在代码中保护特性当前的状态。但我认为这通常不是一个好主意。因为如果代码隐藏在开关之后，很长时间不使用，它会很容易腐烂，继而引起维护的问题。相反，我们应该移除代码以及相应的开关。在版本控制中，老版本总是应该能引用得到。

10.9　总结

有时，工具的特性（例如版本控制中的分支）鼓励我们以伤害其他工程实践的方式使用这些特性。允许开发者在单独的分支上编码，会带来集成和合并的问题。特性开关则让我们推迟决定，继而也会从中获益，还不会伤及持续集成代码这样的工程最佳实践。

不过，特性开关并非过度定制化应用程序的挡箭牌，当我们决定包含某个特定的特性后，就该删除开关，让代码保持整洁。

第 11 章 交付创新

Marc McNeill撰文

创新："引入新事物的行为。"这是字典上对创新一次的解释。Google中也可以查到9200万条和"创新"相关的记录，其热度可见一斑。

但又有多少组织真能做到创新呢？相当多大型企业，甚至是相当具有远见的企业，都无法持续地创新，商业领导者追求的业务创新总是无法兑现其最初的承诺。

我们回到字典对"创新"的定义——"引入新事物的行为"。在芸芸众企业中，许多公司甚至都鲜有作为，更别提"引入新的事物"了。这样的例子比比皆是: 在过去4年里，有家零售银行一直尝试替换掉它们的在线银行产品，但至今仍然没有结果；还有一家传媒组织花了一年的时间来设计其新站点的概念，却没写出一行代码，等等。我们不妨回到12年前，看看Google[①]是如何自我描述的：

"谷歌由Sergey Brin和Larry Page创建于1998年，旨在使人们更便捷地在网络上获取高质量的信息。"

这其中没有谈及浏览器、移动操作系统、文字处理软件、电子表格。Google花了12年的时间，才从一个搜索引擎进化成今天众所周知的Google。

我们再来看看前文提到的那家银行。它们真的适应了这个不断变化的世界吗？它们花费了4年的时间想引入新东西（4年时间几乎相当于Google成立时间的三分之一，几乎等同于Facebook的成立时间），却依然没有产出任何价值。如果同这些领域的人们交流，会发现他们并没有闲着。诸如客户调研、客户分析、视觉设计、商业案例开发等，所有这些事情都是花时间的。

《软件开发沉思录：ThoughtWorks文集》[Inc08]收录了Michael Robinson和Roy Singham的一篇文章，介绍如何走完"商业软件的最后一公里"，从编码完成到最终发布，这条通往产品环境的路上充满了挑战。本文则探讨了产品生命周期的另一端：如何解决IT行业自身的创新问题，以及如何在业务创新中注入敏捷性，并将业务创新融合到交付流程中。

① 使用网站时光倒流机（Wayback Machine），访问http://www.archive.org/就可以看到Google当时对自己的描述。

11.1　价值流向

首先我们要界定一个前提，即只有能够付诸实现的企业创新才能创造价值。现在再看看今天新想法是如何付诸实现的。这要求关注某个新想法变成IT交付项目之前的时间。此外，还需关注它最初是如何成为一个项目的？以及从概念产生到盈利，这条价值链到底是什么样的？

组织逐渐发展，继而出现不同的部门，比如市场部门、产品部门、渠道部门，等等。部门的划分导致各部门只关注各自独立的职责，而忽略其在整体中的位置。每个部门集中于自己的领域里，将部门的产出物当作最终的交付物。

所谓成功是指能及时交付到生产环境中去，但毫不关心交付物能给生产环境的应用程序增加多少价值。因此，开发一款Web应用程序，就有可能出现下面这样状况。

(1) 商业行为始于产品创意的策划与设计。这一阶段的时间花费在调研、市场规模分析和概念开发。

(2) 成立创意机构。该机构交付一些白板设计，这些设计在演示中看上去很不错。通常这时创意机构的使命就已结束。

(3) 业务团队产生受益案例（benefits case，也可以称为business case，业务案例。这一阶段很少要求IT部门全部参与进来，无需提供关于产品价值可靠的评估和成本分析）。

(4) 接下来，业务团队继续开发出更多的文档：其中包括项目启动（Project Initiation，PI）、高层需求、解决方案蓝图和申请资金报价（由于财务方面的复杂问题，项目只能在财年的开始阶段申请资金，这也是实现创新的障碍之一）。

这个过程中的每一个步骤、开发的每一份文档，都是要花时间的，而且涉及多个项目利益方。但时不我待，想想竞争对手此刻正在奋力向市场投入创新产品。而我们却还在检查高层设计、细节设计……我们按照流程进行了几个月，项目才开始步入正规，但实际上连一行代码都没写！

价值流

如何才能让项目利益方相信这个流程缺乏效率呢？如何才能说服他们，告诉他们还有更好的方式呢？首先，要把所有利益方聚集到一起进行研讨，让他们自己看得到当前流程中的问题。这不单单是动动嘴就能说服他们，而是要分析一遍项目的价值流（谁做了什么，花了多长时间，每项活动产生了什么样的价值）。

分析时，选择一个近期的项目，该项目最好始于新的业务创新，止于IT部门发布到生产环境中。同时尽可能多地邀请到项目利益方（注意是参与项目的人，而不是管理项目的人）。然后，

分析所有发生过的活动。每项活动对应一张便利贴。把便利贴贴到白板上，再画上线把它们连起来。指出每项活动的负责人，花费了多长时间，在活动和交接过程中产生过什么误解。很快我们就能得到一张图，可以看到有多少时间花在了增值的任务上（也就是说，完成一些对项目有重大影响的事情），有多少时间在等待，甚至白白浪费掉（比如说，多个小组验收某个非关键的交付物）。这就是对价值流映射的简单解读。这一工具虽简单，却能协助企业找出当前创新能力不足且低效的原因，甚至还能提供一些新思路和新方向。

11.2　新方法

看来，我们的困难在于如何让创新摆脱业务、调研、研发等繁冗阶段，助其真正进入IT生产环境。问题是当涉及IT时，需求都已经从业务部门那边直接丢过来了，而且往往带有简单粗暴的命令口吻："给我做这个。"通过应用价值流分析流程，我们已经发现，在不同的部门之间（业务分析、系统分析、安全等）其实存在明显的交接阶段，这会导致低效和浪费。在新想法能够进入开发和交付引擎之前，总要在商业分析阶段停留过长时间，浪费了大量的时间、精力和资源。

不过纸上谈兵说来容易，如何才能真正避免呢？下面是4条交付创新的小建议。

- 培育一种在整个产品开发生命周期中都重视协作与创新思考的文化。
- 将敏捷（和IT）带入到产品的调研与发现之中，从而引发新观点、新灵感，发展出新概念、新创意。
- 创建一个快速项目启动，尽可能快地分享项目愿景，找到可以交付给用户的最小需求。
- 打造持续设计、持续开发的良性循环。

11.2.1　协作文化

通过使用精益、敏捷和设计思维，我们可以推动业务创新（即创意工厂）更接近IT（即交付引擎）。我们用一种全新、快速、可持续的方法，取代那种让创意一步一步经历各个部门的方法。这种新方法要求IT部门和业务部门从一开始就协同工作。

下面我们来做一个游戏。给你的同事一叠纸，让他把纸放在背后，并撕成两半。你也一样，在背后把另一叠纸撕成两半（游戏规则对你和同事的要求一模一样）。然后我来告诉你这个游戏的含义：如果谁沿着水平方向撕纸，我会给那个人奖励。现在拿出你撕的纸：结果你是沿着垂直方向撕开的。这说明相同的需求，却被解读为两种解释。因此表明语言并不可靠，需要进一步进行解释。而这正是我们要做的。因此我们需要开会，通过会议达成共识。

协作意味着在产品创新的起始阶段就组建一个核心团队。这也算得上是一项社交活动。团队中可能包括开发人员、BA、PM、UX、创意可视化设计人员和业务团队。业务团队由一名产品负责人（单独的、权威的个体，他就好比掌握了全部"真相"，也是项目终极的决策者）和相关

领域的若干专家组成，他们将根据需要投入到项目中去。从产品调研、产生创意、项目启动到产品发布以及后续的工作，他们都要齐心协力在一起工作。

根据以往的经验来看，对于大部分项目来说，花2~4周做调研和启动是合理的。每周开始是都有“启动（kick-off）会议”，计划一周的任务，每周结束时会做“演示”，向项目相关人说明项目的进展。通常，每天都会有一系列研讨会，有些需要全体项目利益方参加，有些只需少数人参加，针对某些具体问题进行深入分析。我们遵循站会（stand-up）、演示等敏捷实践。因此准备一个专门的会议室非常关键；随着研讨会不断进展，会议室的墙壁将成为工作区，布满草图、便利贴和各种产出物。在产品发现阶段，团队会提出一些假设，并进行后续深入观察，验证这些假设。为了检验假设，他们可以走出办公楼，直接观察工作中的用户，或者采访购物中的消费者。他们可以检查当前过程的统计分析信息。也许他们在早上刚有所发现，下午就能回来向团队汇报。团队会把他们的发现记录在便利贴上，这样便于分组。整个团队因此能够发现共同的主题，从而构建下一步的设想，继续验证新的设想；整个过程就这样不断循环往复。这种可视化的方法能帮助团队在考虑“怎样做”之前，快速集中地确定下“做什么”的问题。

从产品概念到实际产出时，团队会使用一些针对需求的可视化模型，比如故事板、草图和线框图（见图11-1）。通过将愿景可视化，团队能够对需求达成一致的理解。可视化模型技术是非常有效的，它可以保证不同的项目利益方“说同样的话”，还能保证他们共享彼此的想法。

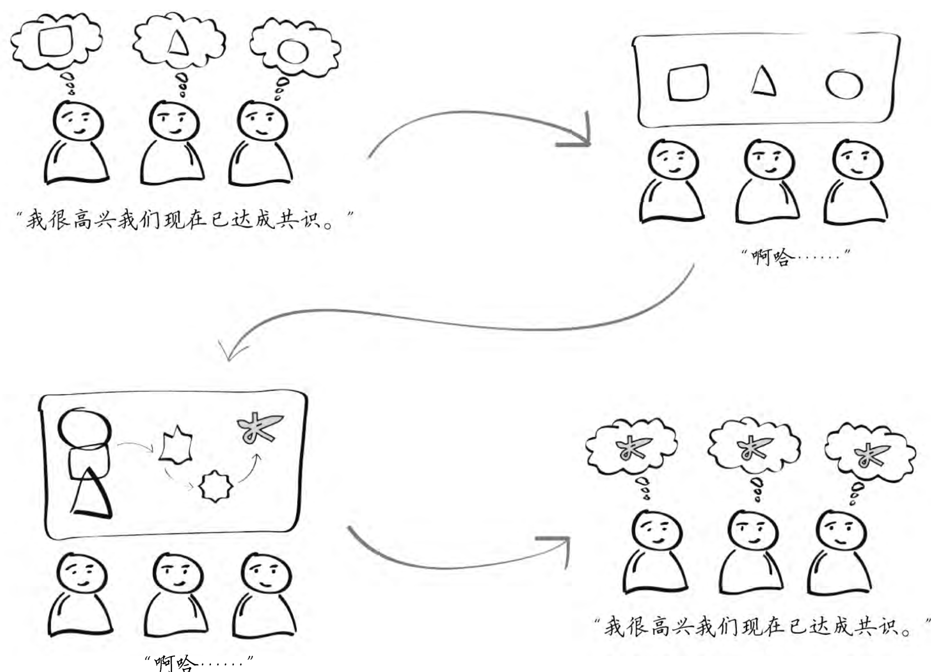


图11-1　使用可视化模型捕获头脑中的创意，分享每个人的想法

11.2.2 敏捷产品调研与发现

想象一下……

想象现在是2007年，苹果还没有像今天这么成功，你也就是个行业新人，正在开发一款和诺基亚的旗舰手机N95抗衡的产品。你是产品经理，负责产品的成败。你满脑子都想着如何打败诺基亚。你让工作的方方面面和N95紧密相连，你对N95的功能倒背如流。你和设计团队开会，他们带来了一些惊人的消息。他们给你列出了新手机的功能。

你：我们直奔主题吧。你们说这款即将投入市场的新手机没有卡槽、没有3G、没有蓝牙、没有高清的摄像头、没有多媒体信息服务、不能播放视频、不能复制粘贴、没有视频摄像头、没有收音机、没有GPS、不能用Java……？

团队：没错，我们放弃了这些功能。我们做的是人们真正想用的手机。

第一代iPhone于2007年6月发布，在此3个月之前，诺基亚的旗舰手机N95发布。只需列表对比一下这两款手机的特性，你就一定会觉得吃惊和失望。做为产品经理，你更愿意参与制作iPhone还是N95呢？

托马斯·爱迪生说过："我要发现这世界需要什么，然后努力发明它。"我们怎样才能发现世界（即消费者）需要什么呢？我们怎样才能知道他们的真正需求呢？我们或许可以将创新产品快速地带入市场，但是如何获知这产品是否正确呢？这时就要去发现需求——找准目标，理解什么是真正的需求。下面是一些用于发现需求的方法。

1. 洞察消费者

我们可以像人类学家一样，研究应用程序使用者的行为，以此理解他们的真正需求。下面这些技巧将有助于这个发现过程。

2. 外面的消费者

走出办公室，观察外面世界的人。不要把这些事只留给项目的用户体验师，每个人都应该走出去进行观察。想一想从线下交互中学到的东西，能应用于在线交互中吗？举例来说，如果我们有一个零售渠道，那么消费者是如何完成交易的？他们会向销售人员询问什么问题？"我知道想要什么"和"我不清楚，请帮我决定"，这是两种不同类型的消费者，他们的处理过程有什么区别？观察他人如何与技术交互可能是一次发人深省的经历，特别是解决方案快要浮出水面时，看到有人使用这一方案，并且纠结于方案中我们视为基础的部分，我们可能就会恍然大悟。

走进客户服务中心，接听消费者的来电。他们在电话里求助什么问题？

“计算机说不”

有一家有线电视公司，该公司把电视节目打成产品包，再起一些华丽的名字。当消费者把电话打到客服中心，想要订阅影片或体育节目时，客服却不可能做到马上办理。他们必须把消费者的需求转化成某种特定的产品，然后再确定应该用哪一种产品包，因为系统是根据业务系统设计的，而不是根据真正的人——消费者的行为设计的。如果能一开始就构造一个优雅的、易用的系统，还需要在客户服务上投入大量的资源吗？

3. 我们的同事到底在做什么

我们不仅仅只为消费者开发产品。那样的话内部用户怎么办呢？有时关键的功能会影响公司的工作流程，在工作中，员工们对此的实际反映如何？让我们探索一下英国的一家大型超市，看看他们的库存管理系统。

价值百万美元的价格标签

每天下班后，员工都会给保质期短的商品（比如三明治）贴上降价标签。他们用手持的扫描仪和带式打印机，扫描商品、打印价签、粘贴价签。整个流程就是这样。但就是比较耗时，每件商品完成这上述3步需要20秒。当面对一货架的商品时，这种工作还是非常繁琐。12件商品需要4分钟。但是如果能用标记笔在“折扣签”上直接写上新的价格，然后覆盖在原来的条码上，就简单多了。完成一个货架也用不了1分钟。

这样做有什么问题吗？事实上，这样节省了3分钟（等待时间）。但还有一个问题。

消费者选好货物后去结帐，不过打折的标签纸却盖住了条码。结帐的员工试图把它撕下来扫描条码，但是却撕不干净。于是，她只能手工输入SKU码和打折后的价格。这样一件商品就花了2分钟，而等待结帐的队伍越来越长。由于“前面只有一人等待”的承诺，超市必须开放新的结帐通道。就这样，原本在价值链末端的一个小问题，在价值链顶端却变成了一个更严重的、开销更大的问题。

（如果你没注意到这个事实，就永远想不到用手持设备为商品打促销标签可不是个可有可无的需求，这其实是个价值百万的需求。）

4. 消费者憎恨你

该死的XX银行！我根本找不到分行的预约电话。**客户服务#Fail**。

这是在Twitter上随机找到的和“客户服务”相关的微博，同时贴上了#Fail标签。我们其实不需要什么社交媒体战略就能听到消费者的声音，消费者会告诉你该如何改进。

5. 培养共鸣

我们并不是总有时间离开办公室去观察人们对产品或品牌的看法（如果是在线品牌，就更不

可能到线下竞争对手的店里观察消费者的购物行为了）。其实，我们还可以培养与消费者/用户的共鸣，以启发思考，开拓思路。让团队通过不同的形式深入到消费者的行为中，消费者使用产品或与服务进行交互时，这种做法可以让我们尽可能多地体会到他们的情绪感觉。

- 如果是手机在线销售，可以到大型购物中心的手机商店，对店员说："你好，我想买个手机。"暂时忘掉关于手机和价格的所有信息，听听他们是如何卖手机的。
- 如果是旅游产品，可以找一家旅行社，告诉他们："想在二月去个暖和干净的地方。"看看旅行社的销售代理是如何指导我们选择飞机、酒店，并代售保险等其他产品的。
- 如果是提供消费贷款的银行，可以试试从不认识的人那里借钱（试一试借钱的感受是怎样的？），或者走进汽车销售处，用信用卡买辆车，感受一下需要贷款的感觉。
- 如果是超市，可以在结帐口呆上一天（在英国，大超市的高级经理在整个圣诞节期间都会呆在店里）。借此感受真正处理购物车的感觉如何？消费者在结帐时会问些什么问题？

6. 创建人物角色

有了对消费者的理解和共鸣，我们就可以创建人物角色了。其实就是一些简笔画肖像，并在画像底下写上他们的角色特征（不是外貌特征）。这些肖像是将要构建的系统的消费者或者用户。这些人物角色如何使用新系统，将直接影响到最初的创意和解决方案。

- 对每类人物角色类型，产品要解决哪些关键需求？如何满足这些需求？满足了这些需求后，对消费者来说就足够了吗？我们需要为产品愿景添加额外的维度吗？
- 如何激发用户使用新产品（或者特性）？换句话说，依靠什么能让用户从知道你的产品并使用产品，同时还要成为产品拥趸？
- 产品将会用于什么环境？比如，相比于在起居室里看电视（身体后仰），早上在火车车厢里看移动设备上的视频（身体前倾）则是完全不同的体验。

7. 技术分析

洞察消费者的过程非常有趣，但是如何利用这一阶段的成果开发出优秀的产品呢？这里就体现出IT部门在整个过程中的价值了。根据逐渐清晰的产品愿景，开发人员要研究当前的技术发展水平，选择相应的技术以解决需求。在这一阶段，很重要的一点是不要被所选的技术架构所束缚。（业务人员通常都反对让IT人员参加创新会议，这也是原因之一。他们注意到IT人员总是以消极的理由如"我们的系统不支持"，阻碍业务上的创新。）而优秀的IT部门应该促进业务人员产生创新的想法，思考潜在的解决方案，并且测试方案的可行性。如果想法真的不可行，再将方案否决。（这个过程是透明进行的：因为IT部了解需求的细节，所以业务人员也能在第一时间获知潜在方案不可行的原因。）

8. 竞争对手分析

成功的创新产品并未必是第一个进入市场的产品。当然这也不是说我们需要重新发明轮子。

其实不妨让其他人去做排头兵，然后我们快速跟进。观察自身领域和其他领域的优秀企业的成功和失败。测试竞争对手产品的可用性，以验证我们的想法。对于一些常见的功能（比如注册、购物车、登录功能），如果有大量优秀实现和设计模式，我们也可以从中汲取灵感。

9. 自省

虽然外界提供了足够多的消费者和灵感。但有时，创新的灵感还可以在组织内部发现。

Code Jam 开发人员有想法

一家国际投资银行的CIO在和我们的一次谈话中十分惊讶。他告诉我们："我手下20%的开发人员都在做开源项目。"这也让他非常好奇，他发现开发人员做这些项目的想法来自于他们在银行的日常工作。由于银行没有给程序员实现这些想法的机会，加上银行的开源策略事实上阻止了开源项目，因此，为了满足自己的愿望，程序员将自己的想法做成银行业务之外的开源项目，这样他们可以真正实现自己的想法。

如何才能将开发人员的才智引导到产品构思与开发的过程上来呢？

这位CIO给出的答案是举行银行内部的Code Jam：将程序员汇聚起来，提出一些业务上的问题，给他们几天时间去自由地发挥创造力，开发各种创新的解决方案。

10. 文档完成

别人的文档不是我们的交付物

ThoughtWorks咨询师和客户的用户体验团队曾会过面，后者当时正在为新项目创建人物角色。前一天，咨询师刚刚见过客户的市场团队，后者与咨询师分享了一些他们的产出物。他们已经创建了适合项目的人物角色。因此咨询师很困惑，他问用户体验团队："为什么不使用市场团队创建的人物角色？""因为那是市场团队的，不是我们的。"用户体验团队如是说。

这是个极端案例。不过，它的确说明了组织乐于重复制造文档和其他产出物。创新固然是要开发新东西，但是有必要从头开始研究吗？我们可以从已有的文档开始，从组织内部其他部门的研究成果开始。在开始任何创新活动之前，我们都应该问清楚之前已经做了什么，并要求团队动用他们的关系在组织中寻求答案。

11. 开发商业案例

组织为项目花费大量的时间和精力开发商业案例。但通常有架构师规划的项目会成为获益案例，而获益案例会从其他从事方案开发的人员减免中得到好处。

业务模式画布[①]（Business Model Canvas）是一个构建业务案例的有用工具。它能展现出业务模型里一些有用的构建单元，这些单元同样可以用便利贴在墙上表现出来，整个发现与探索的过程逐步发展，工具背后的模型也会逐渐显现。

11.2.3 快速启动

前一节介绍的协作可以帮助我们快速地收集背景知识，获得对问题的深入洞察。下一步是将这些转化为愿景、计划、产品。再次强调，速度是关键。只要完成份内的事情即可，让团队向着一个目标前进。

1. 创造性思维

来自IDEO公司的Tim Brown在其TED演讲[②]中给听众做了一个小练习。他要求听众画出坐在他们旁边的人，时间为一分钟。接下来他让大家展示画作，但是大多数人都只是报以“对不起”作为回应。每个人都对展示自己的作品有些拘谨。当谈到创造力时，我们就像回到了童年，会感到害羞，分享创造性的工作成果会觉得不舒服，除非我们觉得它们有些用处。我们担心其他人认为自己的“交付物”不够好、没完成、不够精致。我们不敢设立期望值，担心失败，因此构建起一条验收链，最终由HiPPO（the highest paid person’s opinion，报酬最高者的意见）决定方向。

我相信快速启动过程将会改变这种观念。它是基于协作、创造、游戏以及竞赛的。我们使用*Innovation Games*[Hoh06]、*Gamestorming*[Gra10]中介绍过的游戏，重点在于确定项目的目标和风险。

比如，如果想发现产品最重要的特性或者属性，我们可以玩一款名为*盒子中的产品*的游戏。准备一个早餐麦片盒，在外边糊上白纸，让团队想象他们的产品将放到这个盒子中出售。想象一下如果这个盒子在超市的货架上，我们如何让它看上去更与众不同？销售人员如何向消费者介绍产品？团队分成不同小组，分别对这个盒子进行设计，然后统一向一个小组推销他们的作品。

就风险以及成功条件而言，我们可以把产品想象成一个热气球。燃料能使热气球上升吗？什么样的绳子能够栓住气球？团队成员将这些都写在便利贴上，贴到墙上，将项目风险（栓住热气球的绳子）和燃料（项目成功的关键条件）明确表示出来。

2. 合作设计

我们已经在客户身上花了很多时间；我们已经充分理解了“人”。现在要把关注点转到“做”上。对于每个主要功能，我们都已进行了识别，并演练了高价值、端到端的场景。业务人员、设计者和开发者本着协作的精神，快速地在白板或者纸上拟定出想法，鼓励各方思考：我们期望用

11

① 可在http://www.businessmodelgeneration.com/downloads.php下载。

② http://www.ted.com/talks/tim_brown_on_creativity_and_play.html

户接下来做什么？这个过程刚开始会画一些表示流向的简单方框和箭头，然后做出用户界面的线框草图（通常，正在开发的软件表现为用户界面，所以，由此而生的需求也是很合理的[①]）。的确，为了测试这些想法，可以将这些草稿及时展示给其他团队。不过其他人明白我们想要开发的概念了吗？我们考虑的功能是否可用呢？

团队反复审视，不断修改草图来演进产品的愿景，从而使愿景变得愈加清晰，并分解成用户故事以描述需求。基于合作设计过程的本质，整个团队都要投入其中，捕捉需求变化，做好更改设计的准备。在这个过程中，“什么是真正需要的”和“故事里描述的是什么”之间是不能存在歧义和不一致的。更重要的是，开发人员也在场，因此他们能够理解故事的环境及其背后的意图，还可以对这一过程提出建议，比如如何通过技术提高用户体验（业务人员通常只会基于他们了解的技术实现描述需求，并不一定了解那些能够改进体验的创新技术）。除了功能性需求以外，全体团队还要组织一次会议，确定非功能性需求。这时，开发人员可以在估算和计划之前完成一些技术尝试，验证架构和设计方法是否合理。

3. 故事估算

因为团队分析出了用户故事，因此技术团队也可以对每一个故事提供可行的、高层的设计和估算。由于技术团队也参加了研讨会，参与讨论了业务需求，因此站在了一个更佳的位置上估算每个用户故事。开发人员负责确定出每个故事相对的大小。（在这个过程中，要不断捕获各种假设以验证他们的想法。）完成故事的估算后，接下来要评估团队潜在的生产率。对于一个给定大小的团队来说，一个迭代能完成多少个用户故事？这就需要反复验证，才能获得团队的平均生产率。有了生产率和估算后的用户故事，团队就有了指定发布计划的工具。不过，几乎不可避免的情况是，故事的数量远远超出了时间（或预算）的限制，因此，团队需要讨论他们应该先完成哪些事情。接下来就应该制定优先级了。

4. 优先级

有了前面的估算，整个团队可以制定需求的优先级，以保证先交付价值最高的业务，然后把相关的需求分组，这样就能够向消费者交付一组有意义的功能，其中可能包括引人注目的用户体验，也可能是非常有用的业务功能。在此过程中，常会出现一种比较糟糕的情况，那就是IT部门经常让业务人员产生误解，以为所有低优先级的需求最终就会从交付范畴里剔除出去，从而使得业务人员不太愿意將一些需求标为低优先级。*Innovation Game*[Hoh06]一书中的“购买特性”（buying features）有助于团队理解与所需特性相关的成本。我们在桌子上摆放好估算过的故事卡，按照最终使用者的目标进行归类——记住，从待办事务记录中取出的每一个独立故事并不能组成有意义的产品。每张卡都是有价格（估算）的。然后，我们给产品所有者一些真正的硬币，表示初始概念发布的价值，她要用硬币购买想要的特性。

---

① Jason Furnell是一位ThoughtWorks的同事，他展示过合作设计的过程。详情请看其博客上的视频：http://tinyurl.com/co-design-workshop。

5. 最低可行产品

任何创新要实现价值就必须走向市场。对走入市场的“最小可行产品”达成一致却并非易事。通常，并不是所有人都明白为什么要排优先级。业务人员希望产品的所有特性都能一起实现；否则，为什么还要在一开始就把它们都识别出来呢？为了获得成功，最小可用产品应该满足一组有意义的、连贯一致的、可用的需求，这样才可以转化为业务上的优势。制定优先级的过程并不困难，但说服产品负责人同意逐步发布功能却比较困难。通常，反对声音可能包括以下几类。

我们无法承受负面反馈。尤其是将手机应用发布到应用商店时，常常会听到这种声音。如果产品首次上线就遭到了负面的反馈，确实有些恐怖，这就好像从一开始就宣判了产品死刑。这种担心无可厚非。然而，看看使用者的反馈，就会发现它们基本上都是围绕着对产品已有特性的体验，即产品做错了什么，而不是没做什么。用户常会抱怨产品不好用，充满bug，但不会抱怨产品没有哪些特性。因此，在一个大方向正确的初级产品之上，不断地引入新特性以改善产品，要优于一口气发布一款无所不包的产品。

我们不能发布一个半成品的解决方案。这种担心是合理的，不过这可以通过发布策略予以解决。我们的品牌会有拥趸。我们会拥有愿意帮我们测试新产品的消费者。这是创业公司经常使用的模式：进行初期的封闭beta测试。我们邀请其他人使用新的应用程序，并且告诉他们当前只是beta版本，产品仍然在开发中。这个过程的好处在于，我们在早期就能够收集到基于真实数据的用户反馈。

我们的特性需要和对手相同。这通常源自一种担心——如果新产品未包含同类型产品的全部特性，消费者可能就不会接受。因此，不存在最小可用产品。要克服这种担心，需要识别出消费者最想要什么，给他们交付一个beta产品，然后利用他们的友好和愿望，改进产品特性，帮助他们解决产品使用中的不适或是常见的任务缺陷。如果用户先前没有使用产品的经验，那么向他们介绍产品时，可以考虑下面的策略：对于不知道的产品特性，消费者不会买账。把注意力放到消费者的目标上，不断让客户感到惊喜。与发布消费者不急于需要的功能相比，这种做法更具价值。那些功能改进可以稍后再做。

11.2.4　持续设计，持续交付

在3~4周的时间里，我们将一个模糊的新想法，转化成了经过研究与测试的概念，而且也有了定义好的需求和实现计划。现在，我们可以遵循敏捷方法进行开发了。

产品的成功在很大程度上取决于用户界面的成功，因此，从一开始就持续获得反馈是很有意义的。在早期，构建静态的HTML模板，编写样式，快速实现（丑陋的）JavaScript，产品能够更快速地由消费者测试，在故事开始开发之前，就已经有好几个轮的反馈了（这样的用户测试不必大动干戈，随意进行一些游击式的可用性测试，就会有很多收获——找家咖啡厅，请别人试用我

们的产品即可)。HTML和CSS可以在原型设计和开发团队的代码库之间共享。有人可能对此提出质疑，担心在开发故事之前展示出一个完整的UI，就可能会设置一个无法达到的期望。然而，如果团队中所有项目利益方都以协作的心态一起工作，上述担心就不会成为问题了。更进一步讲，这样做的话，实现阶段的变化会更少，在可用性测试阶段，创新想法在UI上如果对消费者有效，用户在这方面的负面反馈就会少很多。

产品发布只能看做是一次重要的里程碑，并不是最终目标。一旦步入生产环境，就会时时感受到交付的压力。通常的做法是从项目转到业务，相比于此，现在二者的边界已经融合，不像以往那么明显。在生产环境下，我们有用户的产品体验数据；用户的反馈已经变成了“哪些功能不起作用？”“产品应该做什么样的改进？”等诸如此类的问题。设计师需要逐一指定屏幕上元素顺序的日子也一去不复返；分别执行A/B测试或者多版本测试，让不同的用户看到不同版本的产品，这样在决定最佳界面布局或者功能时，就可以由数据驱动了。前文介绍的用于发现创新、合作设计新特性的实践此时仍然有效，最终我们会进入到一种持续设计、持续开发、持续交付的良性循环中。

11.3　总结

大处着眼，小处着手，快速试错，快速调整。

商业领导者追求的业务创新总是无法兑现其最初的承诺。我们不能一味抱怨这是IT团队的问题。敏捷软件开发运动已经证明，IT团队是能够及时响应和交付产品的。本文希望能将消费者驱动的创新和敏捷实践结合起来。基于一个愿景，一个我们为之奋斗的共同图景，我们就能集中精力尽快向消费者发布一个最小可行的产品。通过从消费者那里快速获得反馈，我们就能在投入大量成本之前，来慎重地决定到底是继续开发，还是以很低的成本去中止项目。总之，任何产品研发过程的唯一价值是，将正确的产品交到消费者的手中。

Part 4 第四部分

数据可视化

最后一篇文章探索了一个日益重要的领域：数据可视化，它展示了如何从一个技术产出物创造出引人入胜的可视化效果。

第 12 章

一图胜千言

12

Farooq Ali撰文

如今，数据不再匮乏，匮乏的是对于数据的洞见分析。Twitter每分钟可以发出超过36 000条消息。乐购一天可以产生超过2 000 000条交易记录。在你读完本页之时，YouTube用户已经上传时长超过20小时的在线视频。随着越来越多的公司集成其系统，拥抱语义Web，想要理解所有这些信息变得日益困难。

我们在ThoughtWorks的工作职责，很大一部分就是帮助客户集成、简化以及利用其系统，其中包括系统中的大量数据，我们依赖的工具是信息可视化，或称infovis。在处理这些多得过剩的数据时，信息可视化将扮演着日益重要的角色。图像、文字和数字的有效组合——适宜的组合——能够为数据提供最有意义的表现形式。问题在于，我们如何确定这魔法般的组合？常见误解在于，这个任务最好留给设计师，或是团队里最具美感的人。诚然，当需要创造性思维时，确实需要设计师别出心裁的大胆创意，换言之，要有一种结构化的方式，达成可视化问题的目标，而且形式要遵循功能。在构建更具创新性、更有价值的软件方面，一支采用可视化以及可视化设计流程的团队通常会不负重任。

本章的目标在于，揭开信息可视化的神秘面纱，分享一些能达成设计可视化的结构化思考。我希望本章会对你有所启迪，继而透彻理解infovis，开发出一套共享词汇表，以便与他人讨论，创造出更好的数据呈现方式。

12.1　闻闻咖啡

infovis工作多是由科学和学术研究驱动的。这一点很幸运，因为infovis直接与人的视觉感知相关，而在过去的一个世纪里，科学界已对此做了大量研究。

但不幸的是，同样的说辞并不完全适用于IT产业。因为推动IT产业发展的动力，极少源于相关领域实施的客观研究，相反更多是出现在高尔夫课程上，以及软件供应商和CIO之间的交易中。在以数据为中心的活动中，IT产业已经取得了长足的进步，比如数据的搜集、清理、转换、集成

以及存储等方面，但是以人为中心的数据分析却始终处于落后状态。残酷的现实是，在视觉感知、设计以及有效的可视化沟通方面，以当前研究的水准来看，诸多既有的BI（商业智能）工具还都无法摆上台面。

借用Daniel Pink的名言，科学所知与商业所为之间存在巨大的鸿沟。所以，我们大多屈从于电子表格应用、饼图演示以及分页数据的现状。当然也有例外。当代著名的商业infovis专家Stephen Few说道："有一点我颇为不解，少有像infovis研究这样的情况，它能够解决实际问题，人们却对它一无所知，束之高阁，或许是因为它从未呈现给亟需之人，抑或是它仅以人们无法理解的方式实现。"

但如今，事情出现了转机。infovis并非一个全新的领域。如同人们一直在讲故事一样，以可视化的方式进行沟通信息早就存在。在如今这个数据过载的信息时代，infovis涅槃重生，并以全新且更为深入的方式解决数据过剩的问题。许多行业和组织已经认识到infovis带来的价值，开始用它解决具有挑战性的问题。

当ThoughtWorks对大型IT项目遇到的困难进行评估时，我们首先要运用内部软件质量度量和infovis的最新研究成果，诊断系统当前的"健康"状况。可以想象一下，信息的形式如此复杂——盘根错节的架构、数以万计的代码、经年累月的人为决策与实现历史——因此理清来龙去脉并不容易。这也是我们为何要依赖良好的infovis实践深入挖掘数据，帮助高层决策者作出正确选择的原因。

类似地，根据当前事件揭示出模式与关系，讲述具有洞察力的故事，《纽约时报》就是以此赢得了广泛的赞誉。当今最前沿的一些零售分析公司，比如那些采用忠诚计划的公司，会大量使用infovis，帮助零售商根据消费者的购买习惯及忠诚度，进行"完美"定价、促销以及店铺商品摆放。随着触屏媒体以及无处不在的小屏幕智能设备的普及，我们正被推动着找寻更具创新性的信息可视化方式。

那么，在信息可视化方面，有哪些成功的设计原则呢？

12.2　可视化设计原则

"证据即证据，无论是文字、数字、图像、图表、还是静止或运动，"设计师与infovis专家Edward Tufte如是说，"信息不关心载体是什么，内容也不关心载体是什么。反正这些都是信息。"信息可视化的目标在于帮助我们进行卓有成效地思考，行之有效地分析信息。讨论达成这些目标的方式时，下列原则值得铭记于心。

- **增加信息密度**：并非可视化中的一切皆有其目的。图表，尤其微软自动生成的图表，往往充斥着Tufte这种称为图表垃圾的东西——与信息交流无关的可视化元素。关于这条原则，可以考虑一下如何将信息的油墨比最大化的问题，信息的油墨比就是把表示数据中

有意义信息的油墨（或像素）和总共所用的油墨（或像素）相比较的比率。这方面典型的反例就是3D条形图，无用的立体背景图，冗余的网格线，过度使用的图标。我们应该抵制无关紧要的东西，更清楚地认识如何运用每个像素。

- **利用可视化思维**：早在我们“思考”（也可以说是，聚精会神地处理）可视化信息之前，人们的视觉系统就开始对见到的一切东西进行特征和模式识别。如果信息的呈现方式刚好可以利用预设的视觉处理系统，我们就可以让浏览器更高效地分析信息，从而可以减少“思考”可视化信息的工作。这是以人为本分析的精髓所在，也是本文的关注点。我们会看到一个结构化的可视化设计过程怎样将可视化思维最大化，让我们更卓有成效地分析信息。
- **内容即界面**：可视化思维强调对信息的阅读和消费，但仅有消费是不够的。我们其实还想同数据交互。以人为本的方式对此的应对之道是，创建一个自然的、身临其境的界面。使用iPhone/iPad上的Google地图时，抓取、滑动以及点击地图都非常自然。与地图交互时，这种直接操作和立即反馈，都是“内容即界面”的体现。对于鼠标控制的界面而言，虽然不及前者高效，但这一原则依然重要：比如与环境相关的工具提示、链接高亮、覆盖效果以及动画转换，等等。这一原则的目标在于，淡化工具，让内容成为注意力的焦点。

12.3　可视化设计流程

在大多数情况下，设计可视化流程触及了软件开发价值流的各个方面。到了最后，信息可视化只是这样一个过程：通过代码或者工具，将数据转换成可交互的可视化表现形式。之前已经有人花了些时间定义了一些创建可视化的结构化过程，比如管道模型（Agrawala）、循环模型（Wijk）、嵌套模型（Munzner）。因为这些模型最初是描述在研究论文中，它们可能有些晦涩（至少对于大多数人来说），其实这并无必要，而且这些论文留给我们的印象是，其作者的确很难打动读者。（其中有个人甚至用了微积分！）

抛开那些高深莫测的学术论文，看看图12-1，它很好地展现了可视化设计过程的本质。

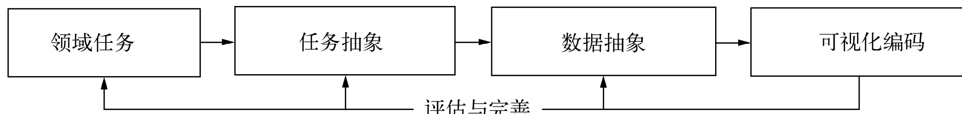


图12-1　可视化设计流程

12.3.1　定义领域任务

好的可视化总是以业务自然领域语言表达出的需求开始。借鉴于敏捷方法论，可以通过用户故事的形势描述需求。比如“作为学校老师，我想了解我班级里的学生表现如何，这样就可以相应地安排中期复习计划。”

12.3.2　任务抽象

很明显，有很多方法可以度量班级里学生的表现。但我可能想知道班级的平均表现，以及对于不同学生、不同科目或是不同的时间段，各个年级的学业表现如何，另外，还可能想知道如果我教课外的知识，谁会逃课。甚至，学生认为最困难的课程也可能是我想关注的内容。此时，你可能会注意到，老师的任务和项目经理、金融分析师等人每天做的事情是很相似的。分析型的任务通常只是一些广为人知的抽象任务，由其特例或混合示例表现出来，这类抽象任务会针对一个或多个度量（指标）。大多数度量（指标）如下。

- **过滤**：找到满足条件的数据。
- **确定极值**：找到包含极限值的数据。
- **查找**：根据一些指标排列数据。
- **确定范围**：找到数据值的区间。
- **发现异常值**：找到包含非期望值的异常数据。
- **特征分布**：确定数据在可用信息范围内如何分布。
- **聚类**：将类似的条目分组。
- **关联**：识别两种信息之间的关系。
- **扫描**：快速查看一组条目。
- **集合操作**：寻找交集、并集等。
- **检索值**：根据某些条件查找特定值。

任务抽象的目标是，将领域任务划分成一系列低层的任务/操作，如果能按照优先级排列更好。我们稍后将看到，能否有效地可视化数据，很大程度上取决于待解决的分析型任务。

12.3.3　数据抽象

能告诉我水温有多少种不同的表示方法吗？我可给你两个提示：热和冷。你可以说水“沸腾了”、“热的”、“温的”、“冷的”或者“冰冻的”，或者你也可以只告诉我华氏或者摄氏的温度值。你如何排列表示冷和热的词？“沸腾了”、“冷的”、“温的”、“热的”、“冰冻的”……哪个词放在第一个呢？不妨试试–1℃、10℃、4℃？我们展示数据的方式深刻地揭示了我们将如何认识和处理数据，尤其是如何将数据可视化。在开始可视化数据之前，我们需要理解每种度量的属性（也就是数据类型）。以下有三个关键的数据类型必须了解。

- **定类数据**：用来决定类型的数据，它本身没有顺序，比如苹果和橘子，或者销售部、工程部、市场部、财会部。
- **定序数据**：用于定义顺序的数据，但本身不表示不同的数值。用户满意度可以用“非常满意”“满意”“一般”“不满意”和“非常不满意”来度量，但它并不能说明两个数据之

间的满意度到底相差多少。同样地，表示赛跑名次的数据是“第一”“第二”“第三”，同样也没有表现出不同名次的成绩差异。

- **定量数据**：数值数据，数值之间的不同是有意义的，比如1cm、10cm和20cm。有时定量数据还分为“定距数据”和“定比数据”，用来表示是否存在一个明确的0点。不过，鉴于这些属于统计学知识，本文不再讨论这些概念。

有时候，我们要根据任务抽象将数据集转换为数据类型。我们可能想知道转换的时机以及转换的原因。下面是两个常见的场景。

- 任务需要对数据集作出一些假设以聚合信息，比如计算平均值和总数。一个敏捷团队在估算用户故事时，可能会用到衬衫的尺寸，然后将尺寸值赋给几何级数（比如S=1、M=2、L=4、XL=8），用以度量项目的范围。
- 有些任务不需要很高的精度就可以高效执行。举例来说，要确定哪些员工没有及时提交工时表，我们不需要知道到他们到底晚了多久。这时完美的设计就自然来了。这也说明了为什么必须明确任务目标。像Apple这样的公司之所以能做到这一点，是因为它们遵循了下面的教诲。

> “设计者知其已达完美，非无可增时，乃无可减时。
>
> ——Antoine de Saint-Exupery

先理解数据类型，然后根据任务从数据集中选择正确的抽象级别（即数据类型），这是有效可视化数据的重要组成步骤。举例来说，下图中的可视化是两种不同的方式——根据不同的城市，表现出公司的品牌影响力（低、中、高）和营收。如果不在左边（A）条形图提供额外的信息，你能猜出哪个城市品牌影响力最低，却营收最高吗？



我们可能会猜是格林维尔，因为它的长度最长，颜色最浅。但其中有几个问题不可忽视。首先，人类天生就会将不同的颜色强度（明亮程度）和定序数据（品牌影响力）关联起来。再加上我们的视觉系统非常善于（在一定范围内）区分颜色的强度。我们的视觉系统也能识别出不同物体之间最小的长度差异，因此“长度”是可视化定量数据的绝佳选择——在本例中就是营收。现在看一看（B）的可视化方案——使用长度表示品牌，使用颜色强度表示营收，这时再完成相同

的可视化任务，即使也可以完成，也困难了很多。

事实上，无论你想从上述数据中得出任何结论，第一种可视化方案总是能给出更好的答案，即使我给第二种加上图例也达不到相同效果。利用可视化的威力快速地感知与任务和数据抽象相关的信息，就能在更少的空间里包含更多的信息，而且还能发现数据中隐含的模式信息，否则大脑的左半球（用于分析且顺序执行的那一半）就必须亲自完成它。这其实是下一步任务——即可视化编码所要完成的事情。

12.3.4　可视化编码

简而言之，可视化编码就是将数据映射到可视化的域（通常是2D的）。高效的可视化编码需要在一开始就对视觉如何运作有一定的理解。与本文其他的主题一样，我无法在有限的篇幅里把这个主题讲得很清楚。不过就以本文的目的而言，下面是我们需要理解的全部内容：视觉感知本质上是一个分为三阶段的过程。

(1) 特性提取。
(2) 模式感知。
(3) 目标导向处理。

在提取特性的第一阶段里，数以百万计的神经元并行工作，检测基本的视觉特性，包括颜色、格式、移动，等等。有一个例子很好地解释了这一点：下图中有多少个“3”？

12904489770478921782372549682584514907601305498
70217144885514907675230876878562468425728189875
41687090010047892178047121536989602439740951287

你花了多少时间找出“3”？答案是5个。在下图中再试一次。

1290448977047892178**3**72549682584514907601**3**05498
702171448855149076752**3**0876878562468425728189875
416870900100478921780471215**3**6989602**4**397409512 87

显然，这次要简单得多。这要感谢我们的视觉系统，它在提取特性的第一阶段里为我们完成了一部分思考工作，也就是所谓的前注意加工。把图型元素弄得亮一点或者暗一点（即改变颜色亮度），也是为了前注意加工而对信息做的处理。因此，颜色强度也称为前注意属性。之所以从第一个图像中找出“3”更加困难，原因在于所有数字都是形状复杂的物体，阻碍了大脑进行前注意加工。Colin Ware是*Information Visualization: Visual Thinking for Design*[War08]一书的作者，

他在书中提出了17个前注意属性。Stephen Few在其著作*Information Dashboard Design*[Few06]一书中，提炼出了最关键的几个属性，如图12-2所示。

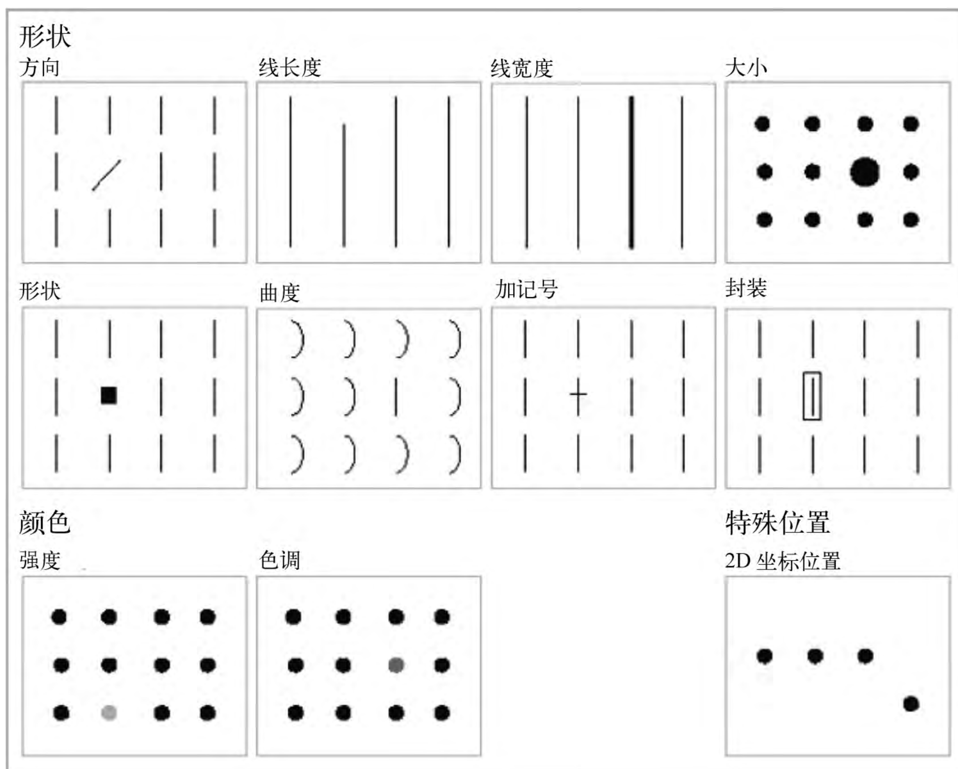


图12-2　前注意属性

在模式感知的第二阶段中，大脑将可视范围分割成明显的区域，然后发现不同区域的结构及其之间的联系。只有在第三阶段中，大脑才会进行真正有意识的处理，执行分析任务。

我们的目标是最大化前两个阶段的作用，让前注意加工祝我们一臂之力，把信息转化成可以由视觉更快处理的形式，这样就能够更高效地理解数据。

1. 提取特性编码

结果证明，根据我们想要编码的数据类型不同，这些属性的有效性会有所差异。那么，我们怎么知道选取哪种数据类型呢？

我找到了Mackinlay排行榜（见图12-3）。它用最简单的方式阐明了相关的概念。这个排行榜还表明了不同数据类型的有效性。

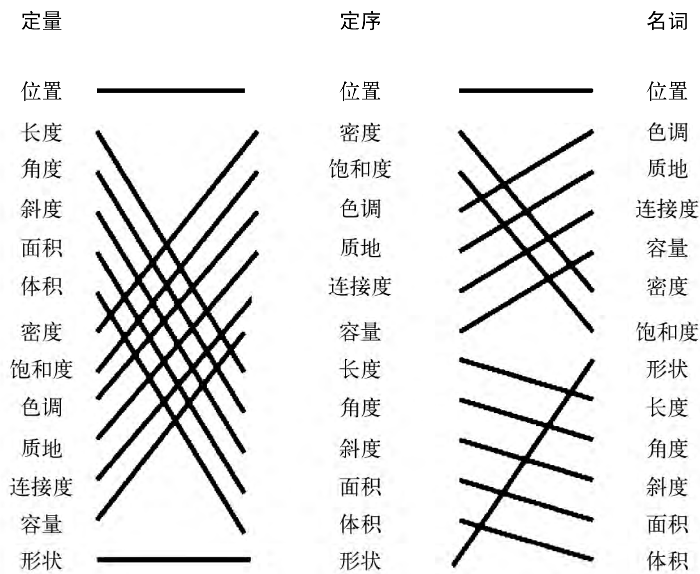


图12-3　Mackinlay排行榜

对于像我这样的初学者来说，这简直就是可视化设计的圣杯。花一分钟时间浏览一下这个排行榜，然后把它和亲身经验关联起来。正如我们所见，三个列表中2D坐标位置排在榜首。这就是为什么传统的*X*-*Y*图像可以有效地表达大量的信息。此外，还要注意一下长度和密度（以前称为颜色亮度）是如何改变定量与定序数据类型的，我们在12.3.3节的例子中用到了它们。

我们可以用这个排行榜来评估一个广为人知的传言：用饼图来比较定量数据是否有效。饼图用面积和角度表达定量数据。然而，根据上面的排行榜，长度和位置胜过了面积和角度。我们自己从图12-4（寻找软件特性的开销）中感受一下。你能说出来构建哪个软件特性的开销最大吗？

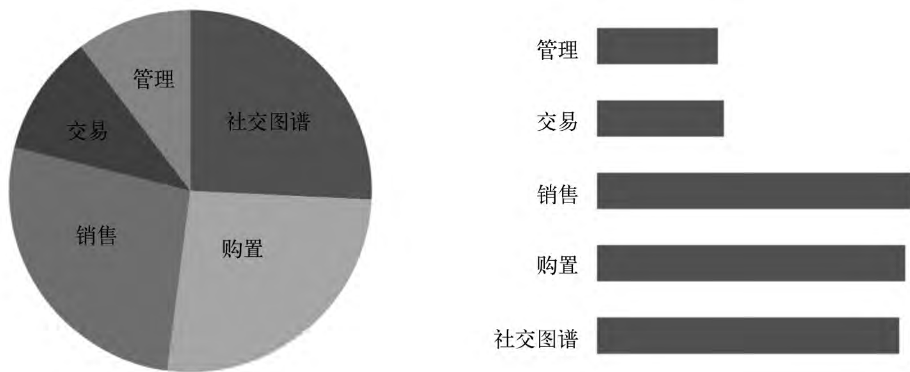


图12-4　寻找软件特性的开销

12

恐怕不能。很明显，使用柱状图能更容易地找到问题的答案。饼图可以说能够有效可视化“局部–整体”的关系，比如我们可以得到一个结论——前三项特性大约各花了25%的开销，但饼图也只能做这么多。如果任务还需要更多的关键信息，比如要对每部分进行比较和排序，就需要对数据做不同形式的编码了。

哪种编码方式最有效，在很大程度上取决于我们的任务。因此，即便是编码排行榜，也应该从这个角度来看，根据不同的任务采用不同的编码方式。同时要记住，我们可以使用多种可视化手段对相同的数据类型进行编码，只要这样能够帮我们更有效地完成任务即可。比如，我们可以将温度的数值用长度、色调（蓝或红）、亮度（从亮到暗）编码。

2. 模式识别的编码

在可视化中，使用视觉系统的第二阶段（模式识别）视觉感知的完形原则（gestalt principle）在对信息进行分组、关联和区分时是非常有用的。比如，我们可能想要引导用户水平地，而非垂直地浏览信息。做到这一点很简单，只要让垂直的空间比水平的空间多一些，浏览者的前注意就会帮助信息分组。这个现象可由邻近性完形原则予以解释：由于事物彼此放得很贴近，因此它们会被认为属于同一组。我们也可以根据封装原则，只利用线和边框对信息进行分组。6种完形原则可以用可视化的方式完美地呈现出来（见图12-5）。

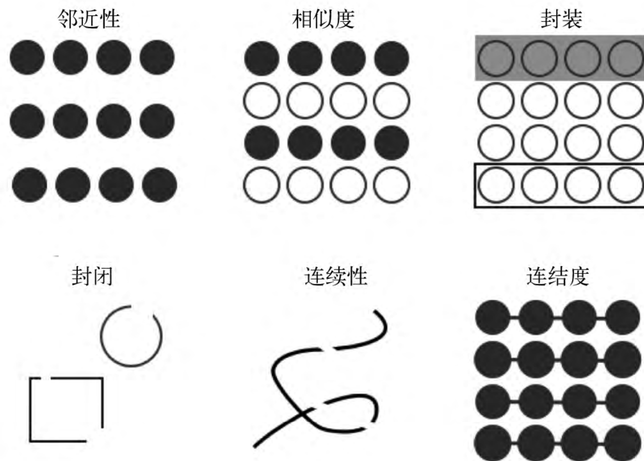


图12-5　模式感知的完形原则

- **邻近性**：我们之所以看到三行黑点，而不是四列黑点，是因为其水平距离更接近。
- **相似度**：我们会将类似的物体看成是同一组的。
- **封装**：我们会将前4个和后4个黑点看成两行，而不是单独的8个黑点。
- **封闭**：我们自动地将方形和圆形连接起来，而不是看到三段不连续的线。
- **连续性**：我们看到一段连续的路径，而不是三段任意的分割的路径。

- **连结度**：我们将连接在一起的黑点归为同一组。

散点图之所以在表现信息的关联性方面如此有效，是因为邻近性、连接性和相似度的定律在发挥作用，让我们可以将信息分组，并填充其空白。图12-6展示的就是散点图中应用完形原则表现相关性，其灵感来自于Hans Rosling著名的TED演讲——“New insights on poverty[①]”。

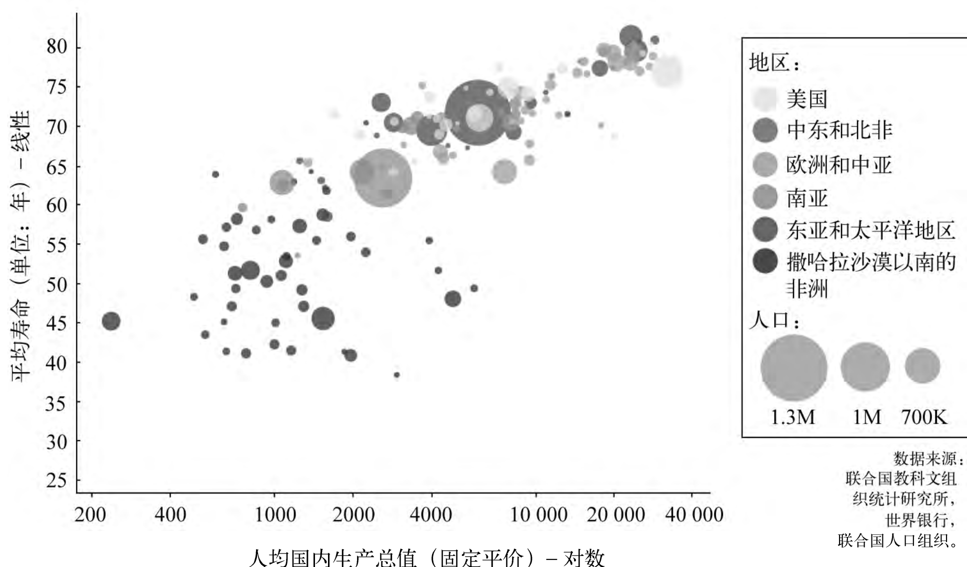


图12-6 散点图中应用完形原则表现相关性

注意，对于我们来说，点的排列本质上是一条线。尽管有些不同的数据有一些“噪音”，但在人均收入和寿命之间建立起关联仍然不难。如果能够把可视化当做一个动态图，关联性将更加明显。如果你还没看过这个演讲，我强烈推荐你看看，然后试试自己是不是也能只凭借数字和单词，就和Hans Rosling一样在演讲中口若悬河。

12.3.5 评估与完善

就像任何有效的软件开发过程一样，反馈周期在这一过程中是至关重要的。本文不会深入讨论软件开发中的反馈过程。不过就infovis来说，可以要求用户参与一些客观测试。请记住下面的建议。

- 利用可视化原型，在所有开发阶段都尽早、持续地收集反馈。不要低估草稿纸测试的力量。不要闭门造车，导致开发出“完美的可视化”，却无法有效地解决领域问题的原型。

① http://www.ted.com/talks/hans_rosling_reveals_new_insights_on_poverty.html

- 度量执行组成领域任务的细粒度任务所花费的时间，对一组同样的信息执行不同的编码方式，这样的测试会很有用。
- 创建一组由不同数据组成的测试场景，检验解决方案在面对不同任务时是否依然有效。
- 即便我们没有讨论可视化数据的质量和真实性，不过，还是要记住数据和度量并不总是正确的。（谎言，该死的谎言，不靠谱的统计数据！）鉴于此，我们不必对数据本身太过认真；项目管理与代码质量的度量就是很好的例子。有时，发现趋势和异常值比追踪绝对数据更重要，而前者正是infovis所擅长的。
- 由于审美偏好不同，所以要明白用户体验中永远有主观性的因素。

12.4 可视化设计模式

现在，我们已对低层的可视化设计过程有所了解，接下来，应该花些时间看一些小例子，了解常见任务的常见可视化手段。在本节标题中，我不严格地使用了“设计模式”一词，这是因为考虑到预先定义了一些编码方式，为常见的任务提供了可重用的可视化框架和数据集，并把余下的部分留给了设计者。

同任何“设计模式”一样，我要提醒你：没有银弹[①]。在使用这些模式之前，必须明确要完成的任务是什么，让任务本身来指导你。当我们只需要一个简单的时间序列图时，在页面上为一个小数据集拍出非常漂亮的水平图就已非常诱人。同样，我们要不停地自问，一张表格是否能帮我们更有效地执行某些任务（比如查询）。

12.4.1 探索随时间变化的数据

我们经常要要查看数据是否会随时间增加、减少还是保持稳定。通常，“直线”可以很好地表现时间，因为它与我们对时间的直觉（无穷无尽）是一致的。时间如果发生变化，可以由向上和向下的角度（前注意加工）展现，这也就帮我们讲数据形象化了。

- **曲线图**：将时间和其他相关变量展现为2D图像上的位置。
- **叠式图**（也称为finger chart）：这种图类似于有多种度量的曲线图，用面积表示度量间的差异。叠式图的一个现实的例子是，寻找价值流中的瓶颈（比如软件开发过程中的分析、开发、QA、业务人员验收）：沿着时间轴，把完成的工作标记出来，再寻找规定时间内完成的面积相对较小的部分。流图是叠式图的一种派生物，其基线（0点）可以按照y轴自由移动。音乐推荐服务Last.fm就使用流图将听众收听的趋势进行可视化。
- **水平图**：如图12-7所示。要可视化大量的时间序列数据时，就应该选择水平图。举例来说，

① “没有银弹”是Fred Brooks在1987年所发表的一篇关于软件工程的经典论文。该论述中强调真正的银弹并不存在，而所谓的银弹则是指没有任何一项技术或方法可以能让软件工程的生产力在十年内提高10倍。——编者注

要想描述30只股票一年的表现，水平图就能派上用场。它用的是时间序列图的形式，同时使用颜色亮度和面积，让我们不必拉伸图的高度，就能观看到更大范围的*y*轴值。颜色有一个有趣的特性，人们通常会过高地估计强烈的、饱和色彩的图型面积。水平图利用这一现象，将更多的信息压缩在更小的空间内，保持图的高度不变。很大的定量值也可以采用不同的颜色，用某条线下的图层面积进行编码。另外，它还将负值反过来放到正值的轴线上，用不同的颜色来表示（比如，红色表示负数，蓝色表示正数）。

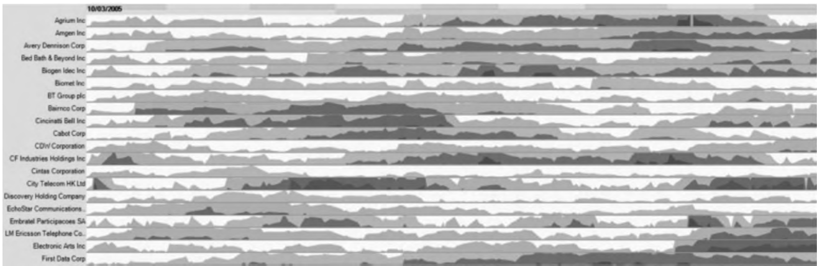


图12-7　水平图（©2012 Panopticon Software AB）

- **折线图**：图12-8所示的就是用于Google Analytics显示面板上的折线图。按照Edward Tufte（知名可视化信息设计专家）的描述，折线图的目标是成为“小巧的、高分辨率的图，嵌入在文字、数字、图像的上下文中”。折线图在最近几年变得非常流行，通常会以小倍数的形式出现。它们和显示面板的关系尤为紧密，比如Google Analytics提供的折线图可以追踪网站流量。

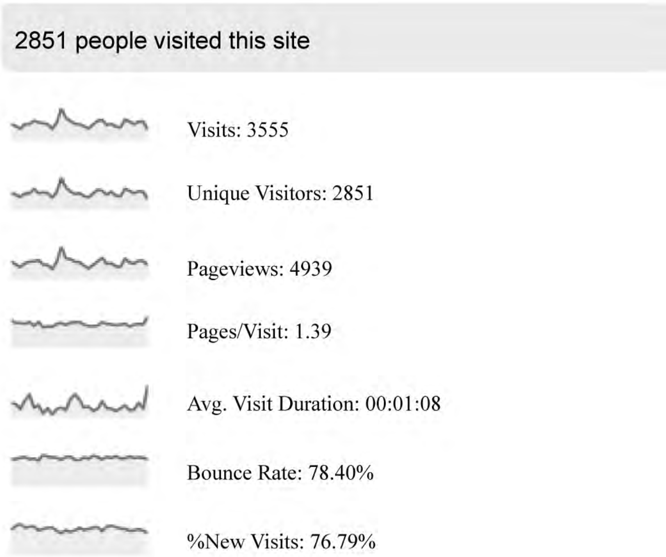


图12-8　Google Analytics显示面板上的折线图

12.4.2　探索相关性

我们并不是总在度量随时间变化的数据。有时，我们需要探索定类和定序数据之间的相关性。这种分析通常会涉及很多条件和变量（即多元数据）。下面是两种常用的模式。

- ❑ **散点图**：图12-6中便有一个示例，即利用完形原则在散点图上绘制数据相关性。出于同样的原因，散点图也非常适合测定离群值（outlier）和异常值（anomaly）。按照最简单的形式，散点图使用2D坐标表现数量值。但是，它可以也为多元值提供更多的选择，比如长度、面积、形状、颜色、外围等，这也解释了为什么说气泡图只是散点图的派生物而已。
- ❑ **矩阵**：矩阵和散点图的形式非常像，不过，矩阵将2D空间分成了网格，以适应定类和定序数据的类型。探索相关性有两种常用的形式，矩阵图表和热度矩阵。典型的矩阵图表用于比较竞争对手产品的特性分析。热度矩阵（图12-9展示了教育绩效的热矩阵）和热度图（图12-13了显示某一天NASDAQ股票变化的热度图）很像，都是用颜色表现某种兴趣（表示为网格里的节点）的数量或顺序。不过与热度图不同的是，热度矩阵更关注建立两组信息之间的关联性，因此节点的2D坐标很关键，比如显示零售公司不同产品线（定型）和不同区域（定型）的盈利能力。

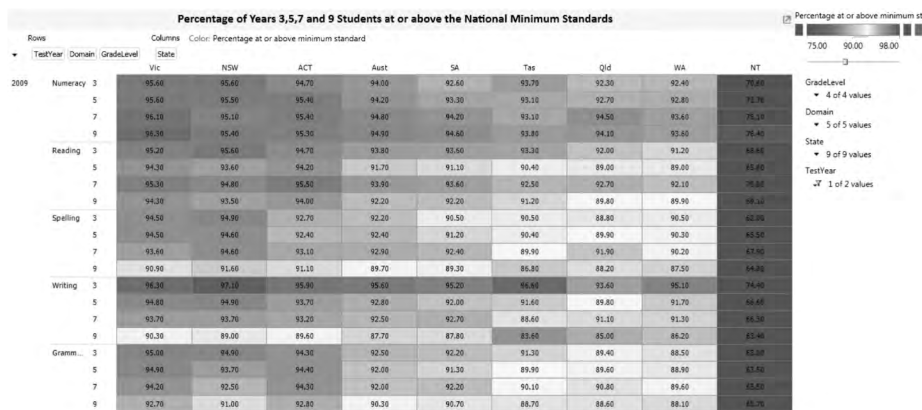


图12-9　按澳大利亚学生所在洲和区域，展示教育绩效的热矩阵（http://tessera.com.au）

12.4.3　探索层次与“局部到整体”关系

我们都善于理解层级以及局部到整体的关系，因为真实世界中充斥着大量的例子。比如，手机电池的电量消耗，其实并不像容器中的液体那样被一点点地排干净；计算机中的目录和文件在硬盘上也不是嵌套地组织在一起的。但是，隐喻让我们和信息打交道时更加轻松。同样，对于某些任务来说，将信息以这种方式可视化格外有帮助，比如归类、发现异常信息、集合操作，等等。除了饼状图以外，下面两种模式也是探索“局部到整体”和层次数据的常用模式：

- **树图**：图12-10是一个树图的例子。树图还有一个有趣的用法，即基于类或目录结构将代码复杂度进行可视化，Panopticode便是如此。用户可以通过鼠标在矩形上的滑动和点击与树图交互，这时树图是非常有效的。每个矩形都代表了一类数据（比如类、文件、目录）。

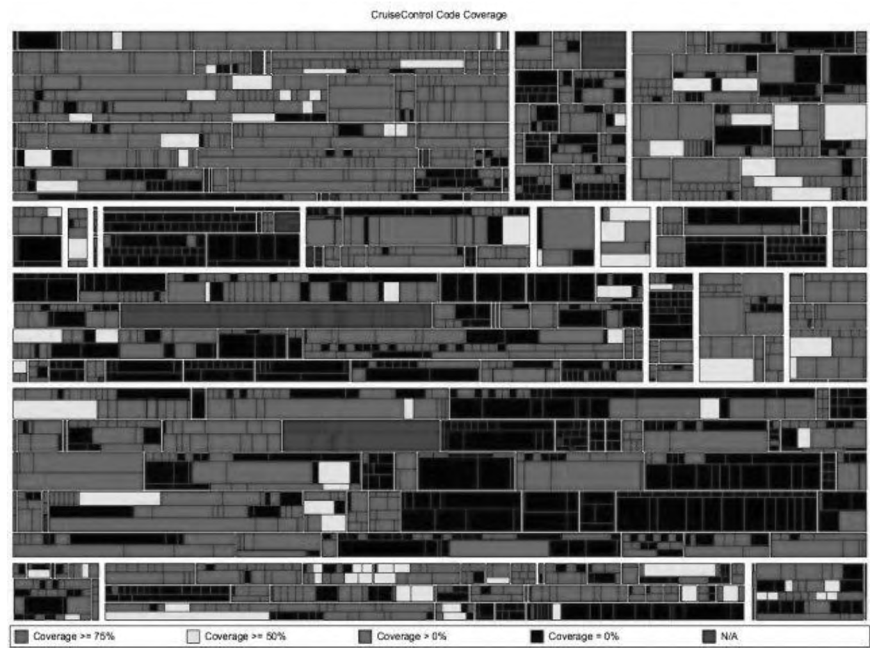


图12-10　树图

- **子弹图**：如图12-11所示，子弹图可以代替仪表盘上的速度表，将量化的“局部到整体”关系可视化，比如KPI。以KPI为例，子弹图将局部和整体的关系表现为不同颜色和亮度的长条，用以表示绩效考核的结果（比如好、满意和坏）。在现实生活中，最接近子弹图的例子是温度计。

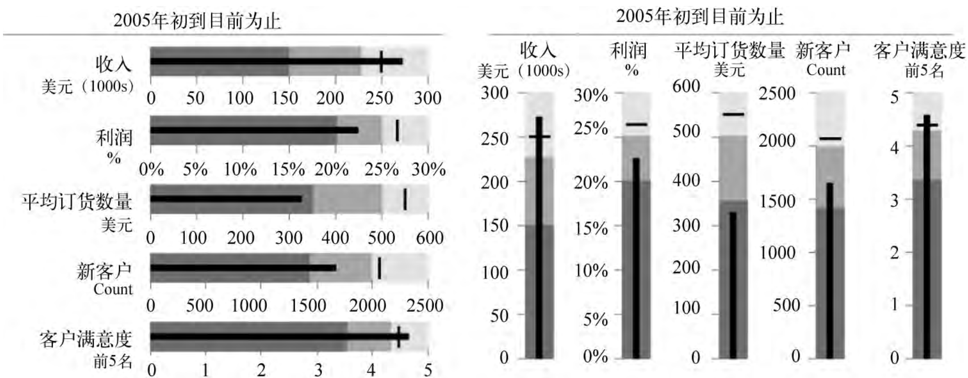


图12-11　子弹图

12.4.4　探索连结和网络

仔细想一想，其实层级和“局部到整体”的关系表现的是两个或更多事物之间一种特殊的连接关系。要将这种任意的互联关系可视化出来（包括层级和“局部到整体”关系），网络图是很好的手段。

- **网络图：**网络图能让我们看出连结关系，这通常是在定型数据间，表现成一组连接节点的线。除了用节点和线来表达连结这一默认选择之外，还有很多构图的方法。循环图可以将关系的扁平列表可视化出来。层级图使用类似于树的结构。多级力导向图用物理和弹簧的启发，对图中节点进行排布。图的种类不胜枚举，不过，选用图形时还是要取决于数据和任务的性质。
- **边绑定：**这项技术能够让网络图更清晰，它的可视化方式是将相邻边绑定在一起，而不是采用两节点的最短线性路径。因此，边邦定很像服务器房间里组织良好的网线布局（见图12-12）。从图中绑定连接增强的线宽和颜色亮度可以看出，这种方式显然非常有效。

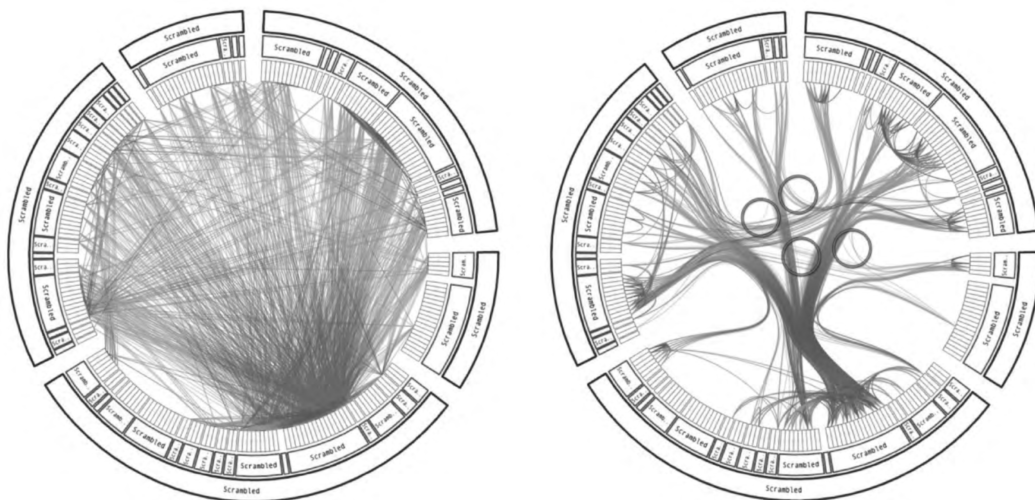


图12-12　边绑定（Danny Holten 2006）

实际上，还有很多有用的模式。比如，我们会发现热度图常用在金融服务领域，显示股票市场的实时活动。如图12-13所示，热度图显示的就是某一天NASDAQ股票的变化。要留意这些模式。只有了解了模式的目标和解构数据的方法，才能在使用不同模式时自行裁剪、修改，以符合自身需要。对于我已经提到过的那些模式而言，有很多工具和框架可以帮我们快速实现它们。



图12-13　显示某一天NASDAQ股票变化的热度图

12.5　工具和框架

实现定制的可视化和前面提到的通用模式，对我们来说正变得不仅越来越重要，而且也越来越简单。我相信，在良好的可视化设计实践过程中，那些能够自己轻松实现可视化视图的人将会扮演关键的角色。

12.5.1　可视化程序库

浏览器提供的用户界面，功能已越来越强大，而且创造这些界面的工具也不断发展。我们现在用HTML 5的画布以及JavaScript所做的事情，曾经只能用Flash和Java applet，才能享受到的附加价值。不过该领域正在发生变化。下面列出了一些程序库。学习它们的最佳方式就是访问其网站，上面都有样例库和示例代码。

- Protovis：该项目由Stanford Visualization Group的成员领导开发，它是一个流行的开源JavaScript图形库。Protovis不仅提供了一套可定制的、支持动画效果的可视化API，它还

支持实现很多常用的自定义视觉效果，包括所有前面谈到的模式。Protovis以优秀的infovis实践和模式为基础，为我们提供了一套声明式的、数据驱动的流畅API。

- **Processing**：它构建于Java平台，是一个成熟的infovis开源程序设计语言。它最初是用于产生Java applets的，不过现在已经移植到其他的语言和平台上，包括JavaScript（Processing.js）和Flash/ActionScript（Processing.as）。
- **Raphaël**：这是一个前途无量的开源JavaScript库，它使用SVG WC3标准以及VML创建图像。与Protovis一样，它拥有标准的基于浏览器的功能和机制（比如DOM操作和实践），可以实现客户端用户接口。GitHub就在使用Raphaël可视化其源代码库的一些度量数据。Raphaël还支持动画。
- **标准图表程序库**：除上述几种程序库之外，还有很多制图、制表库。这些库更关注于提供预定义的可视化功能，所以不如上述程序库灵活。常见例子包括Google Charts（图像和Flash-based）、Fusion Charts、flot（jQuery）以及JavaScript InfoVis Toolkit。

12.5.2　图型化工具

下面谈一谈图型化工具，在我看来，目前通用的infovis工具还不是很多，更多的都是模式驱动的工具，我们可以用它们创建特定的可视化。所以，我们显然无法达到infovis如前文所述那样的灵活性。对于图形化工具，我的经验也很有限，所以只能列出下述几种工具：

- **Tableau**：这是一个灵活且通用的可视化工具，它非常适合前面讨论的infovis设计过程。任何类型的数值都可以用颜色、长度、面积等可视地编码。Tableau还对BI中的数据有很好的支持。
- **Panopticon**：Panopticon能为我们创建很多前面讨论过的可视化，同时还附带一个开发者SDK。事实上，水平图的概念就是Panopticon最先引入的。它支持许多图的创建——热度图、热度矩阵、时间序列图、子弹图等。
- **Many Eyes**：尽管我不在重要的工作中使用Many Eyes，但掌握它能帮我们找到使用infovis的感觉。IBM发明它的目的是希望它成为一款社交应用，用于对上传数据创建和分享可视化。
- **GraphViz**：它是基于文本的工具。随GraphViz一同发布的还有DOT语言，我们可以用它描述性地定义节点和节点间的连接线，创建网络图。

工具和框架领域发展迅速，所以本章内容很快会过时。不过无论使用何种工具或框架，尽量熟练使用那些基本的功能和概念，不要为一些看上去很酷的功能分心，它们通常只是无用的附属品。我自己经常会去除掉很多默认属性，这样就能构建最简单却最管用的可视化了。

12.6　总结

infovis是一个再度兴起且范围广泛的领域，本文不过是浅尝辄止。尽管如此，我仍然希望你

能够感受到infovis的深度，以及如今它给项目、组织带来的重要价值。

如果你是个不想接触可视化设计的人，请记住，世界上还有很多客观的东西也需要设计，比如本文涉及的内容。你不必成为莫奈那样的画家，不必像他那样，能够通过视觉效果，生动地沟通和表达观点。你需要做的只是学习一些客观的设计方法，培养提升主观方面的鉴赏力。对于初学者来说，我强烈建议看一看Stephen Few、Edware Tufte和Colin Ware撰写的书和文章。尝试寻找一个重复发生的分析任务，其中可能牵涉从大量令人烦恼的、包含多个变量的数据中进行筛选，然后给自己一个机会，尝试一下可视化设计。

在21世纪，信息仍会持续增长，速度规模都是前所未有的，尤其是要找到更好的连接不相关信息的方法。从一些现象中，我们已经看到了这个趋势，比如，博客多到无法关注、文章多到无法消化、各种新趋势多到无法跟上步伐、新的市场多到无法抓住。

因此，无论是要理解消费者，为组织做更准确的决策，还是要向公众传达信息，都要记住，我们的任务就是捕获身边的信息，然后将其展现出来。我们其实是在讲故事。因此，考虑到交流的基本前提，不要忘记图片的价值！

参考文献

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2007.

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, Second, 2004.

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman, Reading, MA, 2000.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, Reading, MA, 2002.

[Bur11] Trevor Burnham. *CoffeeScript: Accelerated JavaScript Development*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2011.

[CM08] Wendy Chisholm and Matt May. *Universal Design for Web Applications*. O'Reilly & Associates, Inc., Sebastopol, CA, 2008.

[CT09] Francesco Cesarini and Simon Thompson. *Erlang Programming*. O'Reilly & Associates, Inc., Sebastopol, CA, 2009.

[Coh04] Mike Cohn. *User Stories Applied: For Agile Software Development*. Addison-Wesley Professional, Boston, MA, 2004.

[DMG07] Paul Duvall, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, Reading, MA, 2007.

[FBBO99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Reading, MA, 1999.

[FH11] Michael Fogus and Chris Houser. *The Joy of Clojure*. Manning Publications Co., Greenwich, CT, 2011.

[FPMW04] Steve Freeman, Nat Pryce, Tim Mackinnon, and Joe Walnes. Mock Roles, Not Objects. *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*. :236–246, 2004.

[Few06] Stephen Few. *Information Dashboard Design: The Effective Visual Communication of Data*. O'Reilly & Associates, Inc., Sebastopol, CA, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[Gau02] Hugh G. Gauch Jr. *Scientific Method in Practice*. Cambridge University Press, Cambridge, United Kingdom, 2002.

[Gra10] Dave Gray. *Gamestorming: A Playbook for Innovators, Rulebreakers, and Changemakers*. O'Reilly & Associates, Inc., Sebastopol, CA, 2010.

[HF10] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley, Reading, MA, 2010.

[Hal09] Stuart Halloway. *Programming Clojure*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2009.

[Hoh06] Luke Hohmann. *Innovation Games: Creating Breakthrough Products Through Collaborative Play*. Addison-Wesley Longman, Reading, MA, 2006.

[Inc08] ThoughtWorks Inc. *ThoughtWorks Anthology*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2008.

[Mol09] Ian Molyneaux. *The Art of Application Performance Testing*. O'Reilly & Associates, Inc., Sebastopol, CA, 2009.

[Nyg07] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software*. The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2007.

[OGS08] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell*. O'Reilly & Associates, Inc., Sebastopol, CA, 2008.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, Inc., Mountain View, CA, Second, 2008.

[RWL95] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working With Objects: The OOram Software Engineering Method*. Prentice Hall, Englewood Cliffs, NJ, 1995.

[War08] Colin Ware. *Visual Thinking: for Design*. Morgan Kaufmann Publishers, San Francisco, CA, 2008.

索　引

软件开发与创新

ThoughtWorks文集（续集）

The ThoughtWorks Anthology, Volume 2

More Essays on Software Technology and Innovation

“《软件开发与创新：ThoughtWorks文集（续集）》的一大优点在于主题广泛。技术的快速变化对软件开发者有着很大的影响。本文集不仅涵盖了语言、集成和测试等领域的最新变化，还包括了Java服务器端开发的最新动向。文集既对软件开发新手颇有助益，又能帮助经验丰富的程序员过渡到新的开发领域。”

——Greg Ostravich，CDOT IT专家

“ThoughtWorks的最新文集为我们带来了编程语言、测试和持续交付等领域的最新趋势，同时，又不失实用性。继第一本文集之后，ThoughtWorks又一次将许多具有时效性的、实用的并且引人入胜的文章集结成册，以助软件开发者提升技艺。本文集是所有专业软件开发者的必读书。”

——Peter Bell，General Assembly公司技术副总裁

“ThoughtWorks是一家长久以来令人高山仰止的公司。所以，我欣然受邀，为《软件开发与创新：ThoughtWorks文集（续集）》撰写评论。我尤为喜欢ThoughtWorks精英在实践领域的经验。各位作者的纯熟技艺也在文中展露无疑。更重要的是，该文集所收录的主题与日常的软件开发工作息息相关。在接下来的项目或任务中，我们很可能会从作者的建议中直接获益。

“我坚信，这是本必读书，你一定会喜欢它。”

——Eitan Suez，独立咨询师及演讲家

在软件开发中遇到困难时，如果得知前人也曾至此，便真是幸甚至哉。在本文集中，ThoughtWorks的领域专家们分享自身所学，将他们在IT及软件开发领域中久经考验的最佳洞见结集成册。这些经验会让我们受益良多，从测试到信息可视化，从面向对象到函数式编程，从增量开发到在交付中持续创新，从改善敏捷方法学到顶尖的语言极客范儿。无论何时，当你需要专家建议时，都能从这些已成功解决的问题中汲取营养。

每篇文章都源自一线的实践经验，可以拓展你的技能和视野。无论是从事软件开发、部署、测试的人员，还是软件开发的管理者，都可以从本书中获益。

The Pragmatic Programmers

图灵社区：iTuring.cn

热线：(010)51095186 转 600

分类建议 计算机 / 程序设计

人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-34294-2

定价:45.00元